ISSN 0265-2919

90p 75

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.10 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

A LOOK AHEAD Is man just an evolutionary phase in the path towards a world dominated by intelligent machines? 1481

HARDWARE

PRESSING AN ADVANTAGE We look at Amstrad's disk-based successor to the popular CPC 464 1490

SOFTWARE

SETTING THE SCENE A look at the array tables that hold the data for the cast and props in our interactive drama 1492

OVER THE MOON You can guide your own squad towards success 1500

COMPUTER SCIENCE

ON TWO CONDITIONS We discuss two of FORTH's conditional control structures 1486

JARGON

FROM RELATIONAL DATABASE TO RGBA weekly glossary of computing terms 1489

PROGRAMMING PROJECTS

MAKING A MOVE A routine for our Go game that allows strategic moves 1495

MACHINE CODE

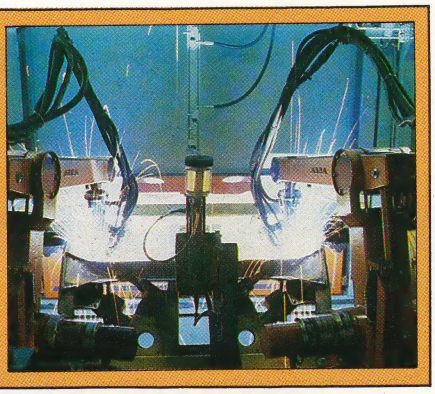
BREAKER, BREAKER How to insert commands into the Spectrum's BASIC 1498

WORKSHOP

INTEGRATING CONVERSION We look at the 7135 integrating chip at the heart of our digital multimeter 1484

Next Week

- **We look beyond electronic computers to optical computers, which run at the speed of light.**
- **Our Spectrum operating system series concludes with an examination of the graphics features available.**
- **Microcomputers are now developing a wide range of applications in industry. We begin a new series on aspects of industrial computing**



QUIZ

1) Although WordStar is designed to run under CP/M, why will it not run satisfactorily on the Amstrad CPC 664?
2) What is the difference between a hierarchical and a relational database?

Answers To Last Week's Quiz

1) Registers are constructed by using bistable 'flip flops' rather than the usual capacitance charge of most memory locations. This produces a faster access time.
2) The array structure is unsuitable for AI inference strategies because the array has to have fixed dimensioning, and would take enormous quantities of memory space to produce a system of any sophistication.
3) A phoneme is one of the sounds that goes to make up the words in a language. Allophones are more comprehensive, as they allow additional accents to be included.
4) The multimeter requires only a single set of control lines, because the driver chip contains the multiplexing logic required to accommodate all of the displays.

Coming Up

- Programming languages yet to be covered in Computer Science include FORTRAN and BCPL.
- A review of the Atari 520ST, a micro from Jack Tramiel — a major combatant in the 'home computer wars'

SPORTS USA We try our hand at games of baseball and basketball INSIDE BACK COVER

PHOTOCOMPOSITION AND RETOUCHING BY HELEN ZAHORODNYJ

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Jon Kaye, Steve Colwill, Steve Cooke, Joe Pritchard, Steve Malone, Steven Vickers, David Mudd; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

A LOOK AHEAD

PHOTOCOMPOSITION AND RETOUCHING BY HELEN ZAHORODNYJ

What The Future Holds
Progress in the development of bio-technology could lead to the disturbing prospect of man-machine hybrids, blending the intuitive processes of the human brain with the logic and mathematical potential of microprocessors. Should this come to pass, we could see the process of human evolution taken out of the hands of nature and placed under the control of man-machines capable of selective self-reproduction

To bring our series on artificial intelligence to a conclusion, we consider the long-term prospect of producing machines with creative intelligence. The effect such machines are likely to have on human evolution is not yet clear, but is certain to have an irreversible impact on future generations of both humans and computers.

By looking on developments in computing as a ladder, we can, broadly speaking, distinguish four rungs of increasing complexity. On the bottom rung we have the mundane bookkeeping tasks that computers do so well. One step up, things get a little more interesting. Here are programs that help people make intelligent decisions — financial forecasting systems and spreadsheets, for instance. These tools have to be flexible, since the user's demands cannot be predicted in advance.

On the third level, we find the application of expertise derived from humans. This is where the bulk of AI applications are concentrated, both now and in the immediate future. Such software is exemplified by the PROSPECTOR expert system, which codified the rules of several geologists. By putting these rules to work in the field, its authors were rewarded with the discovery of a large unknown molybdenum deposit in Washington state. Another one has since been found in Canada.

PROSPECTOR's feats have passed into the mythology of AI, and they are certainly impressive. Nevertheless they do not get us up to the fourth rung of the ladder: to produce machines with creative intelligence. To understand why this is, it is necessary to make a contrast between productivity and creativity.

Productivity depends on following the rules, and computers are excellent rule-followers. They can be more productive than people; but to make them creative has proved exceedingly difficult. To be truly creative, computer programs will have to make up their own rules. Certain workers at the frontiers of AI, however, are trying to make them do just that.

One program that has taken a step onto the fourth rung of the ladder is EURISKO, devised by Doug Lenat of Stanford University. EURISKO is a discovery program we have touched on before in this series. It has been applied in several domains, ranging from a naval wargame to VLSI (very large scale integration) circuit design. EURISKO starts off with a collection of heuristic rules and concepts and applies them to the chosen domains. So much is standard: its novelty lies in the fact that it can modify its own heuristics. This gives the system great power, since it can adapt and specialise its general rules to deal with new situations.

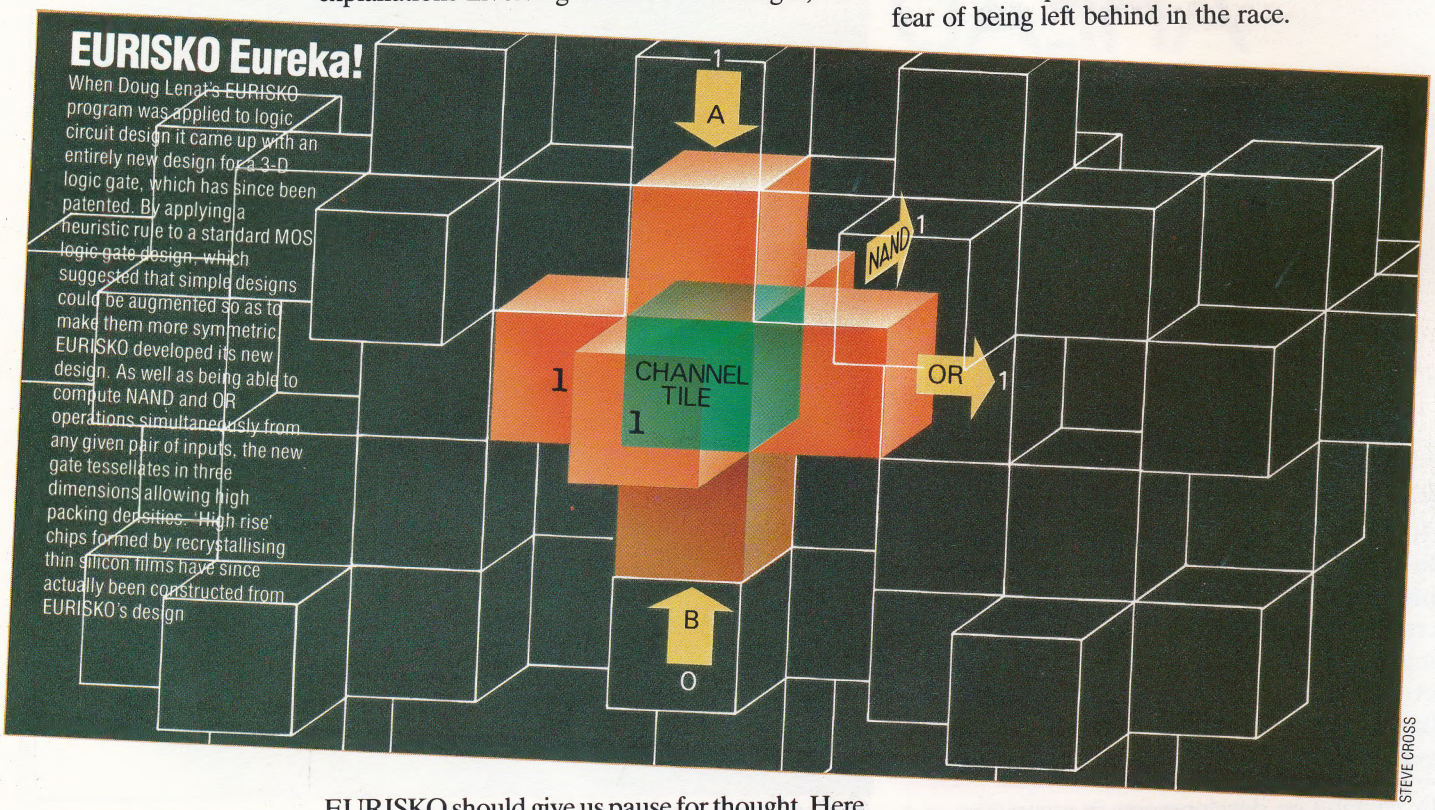
EURISKO has made one discovery for which a patent was subsequently granted, so it can hardly be regarded as trivial. This was a new design for a 3-D logic circuit (a NAND/OR gate), which no one in Silicon Valley — or elsewhere — had thought of. Yet it was generated by the application of a single rule.

When EURISKO was applied to VLSI design it already had a heuristic rule, carried over from previous tasks, that said, in effect: if a concept is interesting, try to make it more symmetrical. Applied to a 2-D device it led to a more symmetrical 3-D version of that device — which has recently been successfully fabricated. A 3-D device, once the manufacturing problems have

been sorted out, can be packed more densely, thus offering greater capacity.

Creativity is seen as the pinnacle of human intelligence, and shrouded in mysterious pseudo-explanations involving intuition and insight; but systems work is firmly rooted in AI research.

Japanese reliance on AI techniques has made the rest of the world scramble back into AI. Governments have been persuaded to approve multi-million pound investment programmes for fear of being left behind in the race.


EURISKO Eureka!
When Doug Lenat's EURISKO program was applied to logic circuit design it came up with an entirely new design for a 3-D logic gate, which has since been patented. By applying a heuristic rule to a standard MOS logic gate design, which suggested that simple designs could be augmented so as to make them more symmetric, EURISKO developed its new design. As well as being able to compute NAND and OR operations simultaneously from any given pair of inputs, the new gate tessellates in three dimensions allowing high packing densities. 'High rise' chips formed by recrystallising thin silicon films have since actually been constructed from EURISKO's design
A
1
NAND
1
OR
1
1
CHANNEL TILE
1
B
0
STEVE CROSS

EURISKO should give us pause for thought. Here is an example of a genuine discovery resulting from the application of a single rule. The creative computer is closer than most people realise.

So far this series has concentrated on the acceptable face of AI. We have viewed AI as the vanguard of computer science — a fertile source of fresh ideas and clever tricks. When these ideas are successful, they filter through to other parts of computing. This has happened to list processing and conversational computing (see page 1361) in the past, and is currently happening to knowledge-based systems. As was pointed out in the first instalment of this series, AI 'exports its successes'.

The medium-term prospects for this type of work are good. We can expect progress on a broad front. There may be setbacks, and some of the bolder predictions may not be realised, but by the end of this century we should have witnessed substantial advances in the fields of computer vision, automatic translation, machine learning and knowledge-based systems. Even the thorny problem of continuous speech understanding should be close to solution.

The Japanese fifth generation initiative has given a tremendous boost to this side of AI. By their ambitious plans, the Japanese have subtly redefined the rules of the game for the computer industry. They expect to produce knowledge information processors in the 1990s which are based on radically new, highly parallel computer architectures. The software that will make such

AI'S DARKER SIDE

There is however what some people see as the darker side of AI. No matter how impressive the advances in AI research, we may not like some of the uses to which they are put — especially if we consider the extent to which AI work is dependent on military funding. More than half the short-term applications of AI are on the battlefield. They include:

- Intelligent submersibles
- 'Smart' munitions
- Cruise missiles
- Self-guided tanks
- Knowledge-based sonar systems
- Homing torpedoes
- Radar imaging systems

and many more that few of us will ever hear about.

Let's just consider for a moment what it is that makes an artillery shell 'smart'. It does not just fall out of the sky and blast a hole in the ground: it selects its target. As it nears the end of its flight it seeks out tank-like objects and adjusts its trajectory to make sure it lands on one. Shoot in the right general direction and you are assured of a direct hit. So that is one immediate application of AI — better ways of destroying tanks.

More alarming are the scenarios some people create from too great a success in the quest for 'ultra-intelligent machines' (UIM) as they have been called — machines with 'super-human'

intelligence. These people envisage a future dominated by UIMs. Without actually 'thinking', the machines will be able to do almost everything that requires reasoned thought better than people can. In mathematics and natural science they will have gone far beyond us. In industry they will so far excel human managers, the scenario continues, that the running of entire economies will be under their control. They may not play tennis very well but they will be better at chess than any human being. In short, we will be their intellectual inferiors. This scenario dismisses as a fallacy the suggestion that we can somehow 'pull the plug' just before machines get too intelligent for us.

THE DISTANT FUTURE

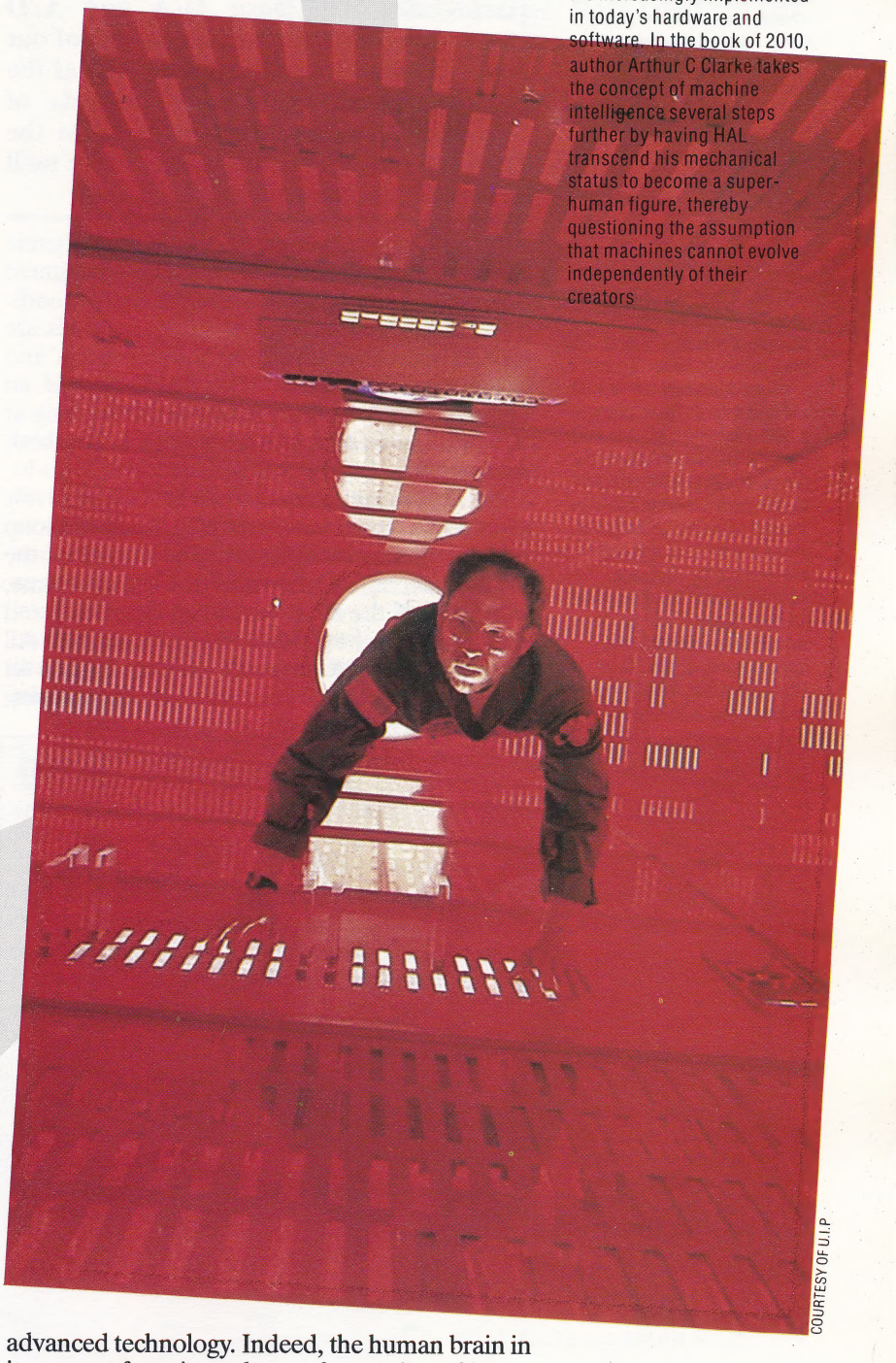
The idea of man-machine hybrids has long been a favourite among science fiction writers, but only recently has it moved from the fantastic into the area of long-term scientific conjecture. The ascendancy of two apparently incompatible areas of scientific research — genetic engineering and knowledge engineering — is largely responsible for this.

Genetic engineering is concerned with the manipulation of genetic code within living material to 'program' genetic improvements into successive generations. Already genetic engineers have perfected techniques that allow them to graft desirable characteristics onto existing micro-organisms in order to produce drugs that were previously prohibitively expensive or impossible to produce on a large scale. It is proposed that, in the near future, genetic engineering methods will be sophisticated enough to produce so-called 'bio-chips'. By programming the actions of the enzymes that divide and reconstitute molecules it will be possible to *grow* logic circuits. The improvement in packing density gained by growing circuits from protein molecules rather then creating them using present day methods would result in a dramatic reduction in size. The idea of living material carrying coded electrical messages is not new: it is the basic mechanism by which the human nervous system works.

At present genetic engineers alter the genetic makeup of wheat to make it more resitant to fungal attack, but these techniques might easily be used to eradicate genetic diseases, such as Down's Syndrome, in humans. Once the first steps have been taken in the manipulation of human genes, further experiments may eventually take place — for example, scientists may attempt to graft a bio-ROM containing a database of information onto the human brain. Although this sounds far-fetched, scientists have already had some success in grafting on electrical apparatus to the brain. A deaf student at Oxford has had the electrical output from a microphone sent into the auditory areas of the brain. After a short period of training she was able to make enough sense of these signals to 'hear' sounds. In short, the brain seems to be capable of learning to interface itself to external electrical sources.

Some people propose that in the distant future our evolutionary descendants may be highly advanced man-machine hybrids of some sort. As the evolutionary process seems to make use of whatever material is at hand — flippers become legs; lungs developed in fish for buoyancy are used to breathe air — it seems reasonable to assume that these hybrid organisms could have at their heart a human brain surrounded by many layers of advanced technology. Indeed, the human brain in its present form is made up of a number of layers that were developed during our evolution.

The uses to which these two new technologies will be put is still very uncertain and whether the hypotheses outlined here become reality seems to depend more on the way society views their desirability rather than their technical feasibility.



COURTESY OF U.I.P

From Machine To Superman
HAL, the computer originally featured in the film 2001 and shown here in a shot from the sequel 2010, starts life equipped with comprehensive speech synthesis abilities and a complex set of rules by which he judges the success of a mission. Such features are increasingly implemented in today's hardware and software. In the book of 2010, author Arthur C Clarke takes the concept of machine intelligence several steps further by having HAL transcend his mechanical status to become a super-human figure, thereby questioning the assumption that machines cannot evolve independently of their creators

INTEGRATING CONVERSION

Having described basic D/A and A/D conversion principles and the design of our digital multimeter, we can now look at the two most commonly used methods of conversion. We also look in detail at the 7135 integrating converter chip, which we'll be using in the construction stage.

Most A/D converters, whether built from discrete components (a rare event these days) or integrated on a single chip, generally use one of two methods. The devices that incorporate these methods are termed 'successive approximation converters' and 'integrating converters'. The 7135 chip is an integrating-type converter, but before looking at its workings in detail, let's briefly look at how both types work.

Successive approximation A/D converters rely on a D/A converter used in a feedback loop together with a comparator. The output of the D/A converter takes place one bit at a time, starting with the MSB (most significant bit) and progressing to the LSB (least significant bit). All the bits in the D/A converter word are initially set to one. As the comparisons are made, the bit being considered is left at one if the D/A converter output is less than the input voltage, and set to zero if the D/A output is greater than the input. After each comparison, the next less significant bit is compared. After all the available bits have been tested (generally eight, 12 or 16), the bits left in the 'one' state allow a current to flow from the D/A converter that matches the analogue input.

An eight-bit conversion takes only eight comparisons, 16-bit conversion takes only 16 comparisons and so on. This makes the successive approximation technique extremely fast — 100,000 conversions per second or more — and therefore ideal for audio, video, radar and other conversions where a lot of analogue data has to be converted to digital form in a very short time. There are, however, numerous disadvantages to this technique, two of which are that successive approximation converters are expensive and require complex circuitry.

INTEGRATING CONVERTERS

When high-speed conversions are not essential (digital voltmeters are a good example), the design choice is usually the integrating converter. Sampling rates of a few readings per second are perfectly adequate if we are not trying to digitise high-frequency AC waveforms.

The output of an integrating A/D converter represents the average value of an analogue input voltage over a fixed period of time. Unlike successive approximation A/D converters, which have to 'sample and hold' the input signal before comparison and conversion can take place, the integrating type digitises the input using time. Clock signals can be used to time the conversion.

Advantages of the integrating converter technique, which will be used in our design, include high-accuracy, non-critical components (apart from the reference voltage components), excellent noise rejection, no need for inherently difficult 'sample and hold' circuitry and comparatively low component costs.

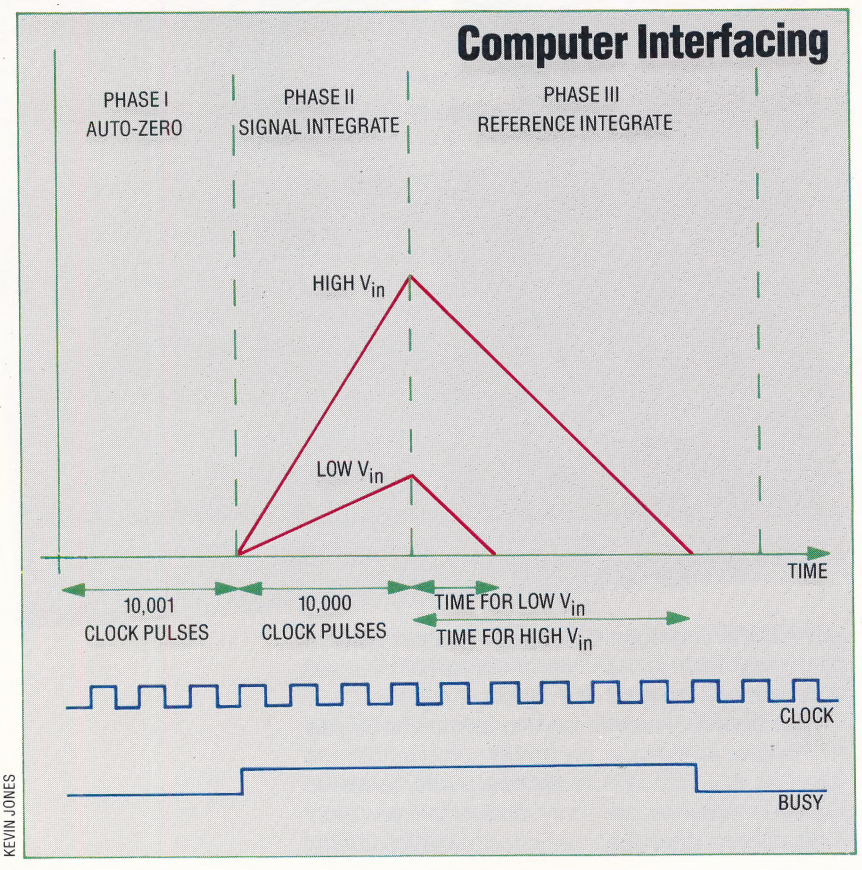
In a typical integrating A/D converter circuit, conversion takes place in three phases. These are known as the *auto-zero* phrase, The *signal integrate* phase, and the *reference integrate* phase. They are, therefore, relatively immune to the effects of high-frequency fluctuations in input 'noise' (unlike 'sample and hold' successive approximation converters).

All that is required for an integrating A/D converter, apart from a stable clock signal, is an accurate reference voltage. This is easily supplied, as we will see later, using commonly available band gap reference diodes. These diodes limit the

Sloping Towards The Busy Pin

Phases I and II of the conversion process are fixed in length, but the time taken to perform the third phase is proportional to the size of the original input voltage — V_{in}. During this phase the previously integrated input voltage is reduced uniformly to zero: the larger the initial voltage, the longer this process takes. This fact enables us to interface our DVM to a computer.

The BUSY pin of the 7135 A/D converter chip goes high at the start of the signal integrate phase and stays high until one clock pulse after the output from the integrator crosses zero. If the BUSY signal is ANDed via a logic gate with the CLOCK signal then the output will take the form of valid clock pulses — that is, clock pulses between the start of the second phase and the end of the third. These valid pulses can easily be transmitted via a serial link to a computer. After subtracting 10,001 clock pulses to take account of the signal integrate phase and the final null pulse at the end of the reference integrate phase, the remaining number of pulses will be directly proportional to the original input voltage. Thus, after calibration, 20,000 pulses corresponds to 2v, and so on



KEVIN JONES

maximum voltage that can be applied.

Most integrating A/D converters, including the 7135, use the so-called 'dual-slope' technique. The three phases operate as follows:

Phase 1, Auto-Zero: Analogue component errors are nulled by grounding the input and storing error information in an auto-zero capacitor.

Phase 2, Signal Integrate: The input signal is integrated for a certain number of clock pulses — 10,000 clock pulses is typical for a 4.5 digit converter. When the integration period is over, the voltage produced is directly proportional to the input signal.

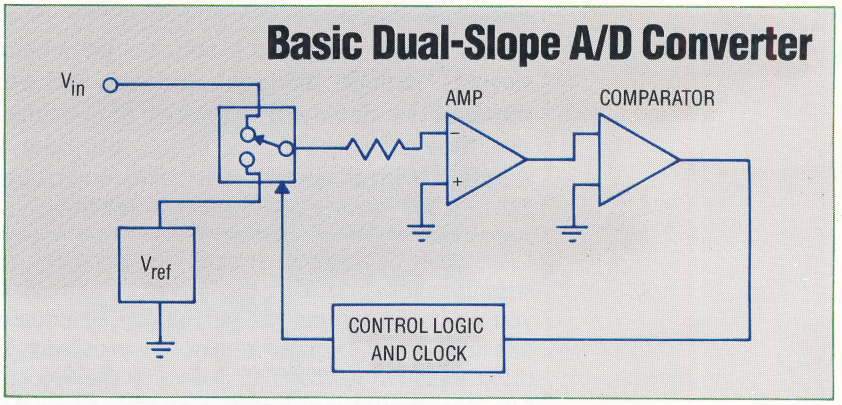
Phase 3, Reference Integrate: At the start of the phase, the input to the integrator is switched from the input voltage to be measured to the reference voltage. The number of clock pulses counted from the start of the reference integrate phase to the time when the output of the integrator passes through zero represents the magnitude of the input signal.

Dual-slope A/D converters are inherently accurate as the whole process depends only on the absolute accuracy of the reference voltage (hence the reference zenner diode must be chosen with care) and the quality of the clock pulses. The clock does not need to be of any particular frequency and it is not necessary for each pulse to be exactly the same duration. One of the reasons we have chosen to use the 555 timer chip (see page 1464) for the clock is precisely because it is a basically stable and accurate device.

The stability of other components, such as the integration capacitor, are of no consequence so long as they do not change in value during the course of any one conversion cycle ('successive approximation' converters, on the other hand, depend for their accuracy on the precise matching of the resistors in a resistor 'ladder'). Since it is easy to keep the accuracy of the clock stable to better than one part in a million, dual-slope A/D converters have every advantage over successive approximation converters so long as high conversion speed is not required.

THE 7135 CHIP

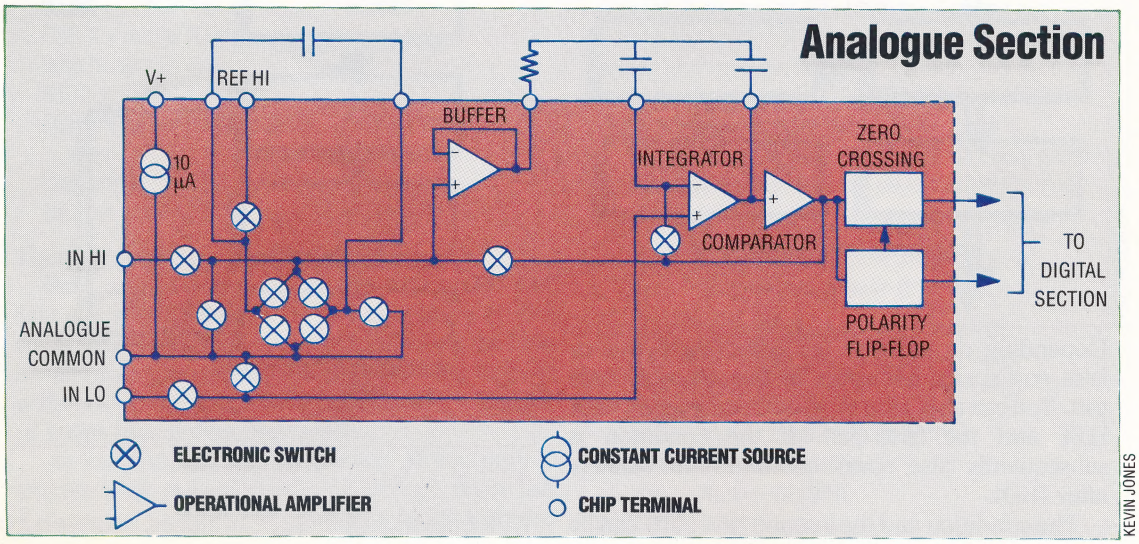
The illustration shows the analogue section of the 7135. Most of the symbols will probably be familiar except, perhaps, the double circle and the circles with crosses in them. These represent constant current sources and analogue switches, respectively.



Around The Circuit
An integrating dual-slope A/D converter performs its task in three distinct phases: auto-zero, signal integrate and reference integrate. During the signal integrate phase the circuit switches in the input voltage — V_{in} — and during the reference integrate phase the reference voltage used to calibrate the converter — V_{ref} — is switched in. The entire operation is coordinated by an external clock source

During the three phases of the converter's action — auto-zero, signal integrate and reference integrate — it is important to note that the number of clock pulses against time is fixed for the first two and variable for the last. Auto-zero takes 10,001 clock pulses, signal integrate takes 10,000 pulses, and reference integrate takes as many clock pulses as are needed for the output of the integrator to pass zero (up to a maximum of 20,001). This enables a very simple technique to be used to read the A/D output by computer, using a serial interface.

The frequency of the clock, within broad limits, does not matter. Clock frequencies as low as 5KHz will work (giving a measuring cycle of around 10 seconds), or may be as high as 1MHz. However, very low clock rates cause errors through leakage in the reference capacitors and very high clock rates require tricky compensation of the integrating capacitor using carefully matched resistors. Suffice it to say that any clock frequency between 100KHz and 160KHz will work admirably.



Analogue Passage
The diagram shows the layout of the main analogue components present on the 7135 A/D converter chip. Components within the shaded area are actually 'on-chip'. All other components are external. The analogue section of the 7135 is essentially an integrating dual-slope A/D converter circuit providing zero crossing and polarity signals to the digital section of the chip. In conjunction with external clock signals these are sufficient to provide the digital section with the information it needs to generate the chip's digital BCD output

ON TWO CONDITIONS

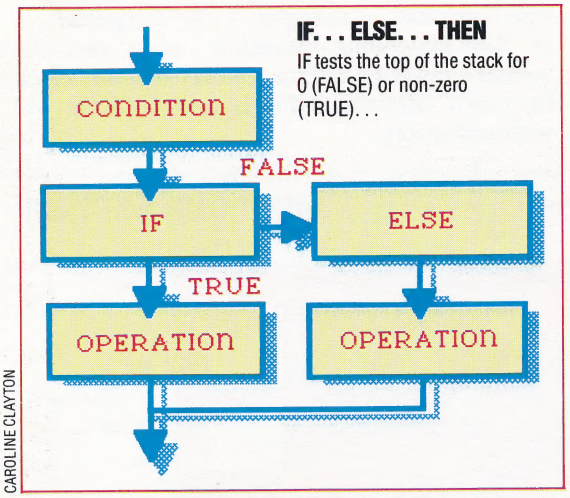
So far in our series on FORTH we have seen simple, 'straight through' routines, but not much of the variety of structures that control the flow of a program. Here, we take a detailed look at several structures, including IF . . . THEN . . . ELSE and the DO loop.

Like other languages, FORTH has a variety of structures for controlling the flow of a program. If you are used to languages with control structures, like PASCAL or C, or more advanced BASICs, such as those included in the BBC Micro or the Sinclair QL, you should easily recognise what these do. Just remember, however, that because of the way FORTH uses a stack, the structures will often seem to be written backwards.

Another point to keep in mind is that these structures can be used only inside colon definitions. This is because branch instructions need to be compiled by FORTH into conditional and absolute jumps. This process requires a certain amount of time, which would not be available if the instructions had to be compiled during program execution. By inserting the structures within colon definitions, you give FORTH a chance to work out exactly what is going on while the definition is still being incorporated into the dictionary.

Here is what is available in standard FORTH:

●condition IF true part ELSE false part THEN



Depending on whether the condition is true or false, FORTH executes either the true part or the false part. You can also, as in most other languages, omit ELSE and the false part. If the condition is subsequently false, FORTH continues immediately after THEN.

The condition and the true and false parts can be any sequence of FORTH words, possibly with more IF. . .ELSE. . .THENs. The condition must be placed onto the stack, and IF will then remove it. Here is an example, which takes a number off the stack and prints whether it is odd or even:

```
: PARITY        ( n--)
                (prints n "is even" or "is odd")
   DUP .        (prints n itself)
   ." is "
   2 MOD 0 = IF
      ." even"
   ELSE
      ." odd"
   THEN
;
```

MOD gives the remainder when the first number is divided by the second.

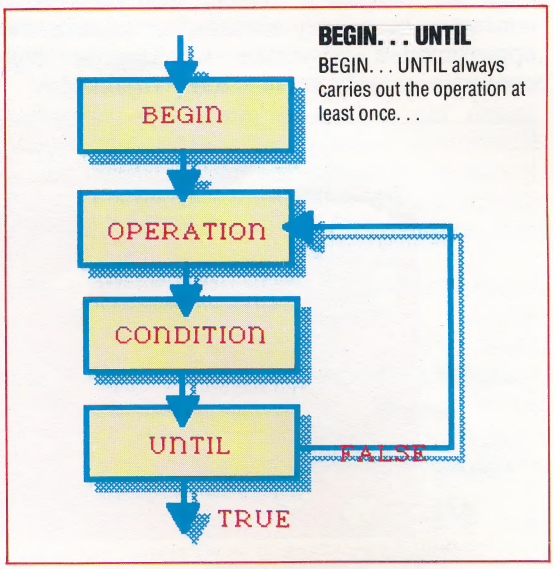
●BEGIN. . . UNTIL

This is for looping until a condition holds. Most modern computer languages, including the newer dialects of BASIC, have something similar. Its format is:

BEGIN loop part condition UNTIL

and it executes the loop part and the condition; the condition is just the end of the loop part that leaves something in the stack for UNTIL. For this reason, the loop part must be executed at least once. For example:

```
:  2POWERS (--)
                (prints all powers of 2 less than
                10000)
   1            ( initial power of 2)
   BEGIN
      CR DUP. (print power of 2 on a new line)
      2* (get next power of 2)
   DUP 10000 > UNTIL (test whether too big yet)
   DROP (drop last power of 2)
;
```



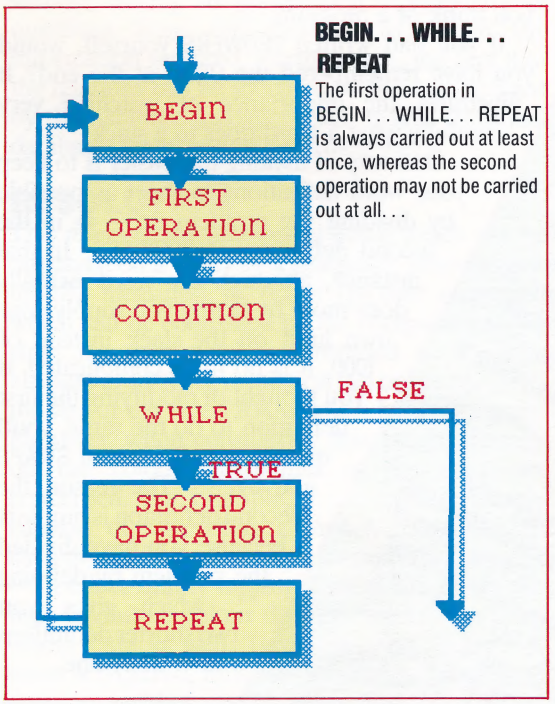
●BEGIN. . . WHILE. . . REPEAT

WHILE has slightly more possibilities, because it lets

you decide in the middle of the loop part whether you want to stop looping, not just at the end. Its format is:

```
BEGIN 1st loop part condition WHILE 2nd loop part
  REPEAT
```

Like the loop part with UNTIL, the 1st loop part here is always executed at least once and it finishes with the condition left on the stack. But now there are two important differences from UNTIL. First, looping now stops when the condition is false. When looping stops, FORTH skips the 2nd loop part and carries on after REPEAT. Secondly, if looping is to continue, FORTH executes the 2nd loop part before looping back to BEGIN.



BEGIN. . . WHILE. . . REPEAT

The first operation in BEGIN. . . WHILE. . . REPEAT is always carried out at least once, whereas the second operation may not be carried out at all. . .

● DO. . . LOOP

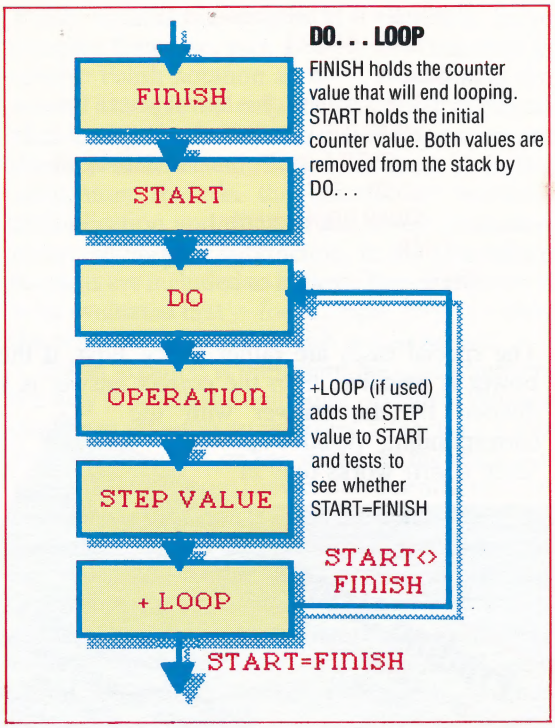
The DO loop in FORTH does the same job as the FOR loop in BASIC and many other languages. Let's compare it with the BASIC implementations:

BASIC	FORTH
FOR X = initial TO limit	(limit+1) initial DO
loop body	loop body
NEXT X	LOOP
FOR X = initial TO limit STEP step	(limit+1) initial DO
loop body	loop body
NEXT X	step +LOOP

There are clearly a number of differences.

First, as you might expect, the initial value and limit precede DO, so that DO can take them off the stack. The limit is first. Less obvious is that the limit is one more than the final value you want to loop with — so 4 0 DO loops with the values 0, 1, 2 and 3 (it loops four times), but not 4.

As in BASIC, the step is optional. If you omit it and use LOOP, the step is taken as being 1. If you use it, then you need +LOOP instead of LOOP, with the step just before +LOOP.



DO. . . LOOP

FINISH holds the counter value that will end looping. START holds the initial counter value. Both values are removed from the stack by DO. . .

+LOOP (if used) adds the STEP value to START and tests to see whether START=FINISH

In BASIC, there is always a control variable (X in the previous example). In FORTH, you don't use a variable like this; instead, there is a FORTH word I that puts the loop value on the stack. If you have one DO loop inside another, then I refers to the innermost loop value and J refers to the next one out. If there are more DO loops outside that, then their loop values are still in existence, and being counted correctly.

Here is an example that raises one number to the power of another. Unlike BASIC, FORTH hasn't got such an operator built in; you define your own:

```
:  ** (m,n -- m**n)
  DUP 0 < IF (if n <0 answer is taken to be 0)
    DROP DROP 0
  ELSE
```

Conditions

The following words are useful for working out conditions for IF, UNTIL and WHILE. Their results, which they leave on the stack, are either –1 for true or 0 for false, but only in FORTH-83. Older FORTHs leave 1 for true or 0 for false.

IF, UNTIL and WHILE don't insist on being given 0, 1 or –1; 0 is false and any other number is true.

=	m,n -- true or false (true if m = n)
<	m,n -- true or false (true if m< n)
>	m,n,-- true or false (true if m> n)
0=	m -- true or false (true if m = 0)
0<	m -- true or false (true if m < 0)
0>	m -- true or false (true if m > 0)
NOT	true or false -- false or true
AND	m,n -- m AND n
OR	m,n -- m OR n

```
    DUP 0 = IF (if n=0 answer is 1)
      DROP DROP 1
    ELSE
      1 (will multiply this by m n times)
      SWAP 0 DO ( to do loop n times )
        OVER * (multiply top of stack by m)
      LOOP
      SWAP DROP (drop m)
    THEN
  THEN
;
```

The special cases are rather tricky. First, if the power is negative, then the correct answer is 1 divided by the answer with the corresponding positive power. Since FORTH works only in integers, you can't do this division exactly, so it's sensible to say that ** returns 0 as its result. The special case when n=0 is more subtle. On the face of it, if you multiply 1 by m zero times (by going round the loop zero times) then you should get the result 1 in any case. However, a FORTH DO loop is always executed at least once and this spoils the case n=0. In fact, there may be times when you will not necessarily want the DO loop executed in a program, in which case you should take special precautions.

Here is another example. It is a more thorough version of 2POWERS, which we gave earlier. It uses not only I, the looping value, but also the word LEAVE, which stops the looping immediately and carries on after LOOP or +LOOP:

```
: 2POWERS     ( n -- )
( displays indexes and powers of 2
so long as the powers are less than
n)
15 0 DO (2**14 is the biggest that can be printed)
  DUP 2 I ** < IF ( if 2**I>n)
    LEAVE
  ELSE
    CR I . 2 I ** .
  THEN
LOOP
DROP ( drops n)
;
```

With these programming structures, you can do without GOTO and line numbers — in fact, FORTH simply hasn't got them. You may find this takes a little getting used to after using BASIC, but the fact is that line numbers are something imposed on you by BASIC — they are not an intrinsic part of the way you think of a program.

If you had written 2POWERS yourself, would you have remembered the DROP at the end? It illustrates the importance of watching very carefully what a word does to a stack.

One way of making this easier is to keep your word definitions as short as possible by dividing the problem up — as in the second definition of 2POWERS. In that instance, although the word actually does more by letting you supply your own limit on the stack instead of 1000, it is no more complicated. If you thought of modifying the first definition to do the same, you'd quickly get all sorts of SWAPS and OVERS. The reason the second definition is simple is because you have divided the problem by defining **, which gives you two shorter definitions instead of one.

Printing Strings

To print a string from within a colon definition, you use ." in the form:

." string"

You need at least one space after ." because it is a word. If you have more than one, then the spaces after the first are considered part of the string. The string follows ." (so it is not reverse Polish notation — see page 1475) because the FORTH stack cannot handle strings. String variables are, in fact, not supplied as standard in FORTH, but there are ways of extending the language to handle them. The word CR (carriage return) sends a new line to the screen

Limitations

Although the main ideas behind DO are fairly straightforward, there are some oddities connected with it. First, if the step (before +LOOP) is negative, the loop value may actually reach the limit instead of stopping just before it. Thus 2 0 DO . . . -1 +LOOP loops through the values 0, -1 and -2

Secondly, we saw how a DO loop is always executed at least once, even for 0 0 DO . . . LOOP where you might want it to be skipped over. FORTH-83 will surprise you by executing the loop 65,536 times. -1 0 DO . . . LOOP will do it 65,535 times, -2 0 DO . . . LOOP will do it 65,534 times, and so on.

The simpler answer is not to do these things by accident, but it makes sense if you are aware of the way computer integers can be treated as either unsigned or signed (two's complement — see page 1168). If the initial value is already past the limit, then it will go up past 32,768, which it treats as equivalent to -32,768, and then move up through the negative integers so that it reaches the limit from below.

With older FORTHs, if the initial value is already past the limit, the loop is performed just once. Older FORTHs also treat LEAVE slightly differently. They don't jump straight out of the loop, but ensure that an exit will be made the next time LOOP or +LOOP is encountered

RELATIONAL DATABASE

This is a database where the record fields are connected together by mathematical relations (see page 1468). This is in contrast to an hierarchical database in which the individual fields are considered a subset of a record, and you must first access the record before you can examine the individual fields within it.

The flexibility of the *relational database* has made it increasingly popular as a method of producing databases as opposed to the hierarchical system. This is because it is not only faster, being able to interrogate the required fields directly, but is also considered more 'user friendly' in as much as the user can ask a question directly, without having to go through a hierarchy of questions from the database to obtain the information required. Furthermore, the fact that the relations can be specified by the user and not by the database manager itself, makes relational databases far more flexible in their use.

RESOLUTION

This refers to the amount of detail which can be displayed on a VDU by a computer. *Resolution* is usually measured in pixels (picture elements) which can be controlled by the computer's video chip. As each pixel is controlled by a single element in an array, resolution can be measured by the size of the video RAM which holds the array. However, when considering this, much depends on the colour capabilities of the computer concerned. For example, if a computer is only capable of supporting a monochrome display, each pixel will require only a single bit to control it — in effect simply turning it on or off. However, high-resolution colour displays need to store the colour data as well. Therefore a whole byte or more is required, thus multiplying the amount of memory needed several times. For example, on the BBC Micro, `MODE 0` is a two-colour display with a 640 by 256 resolution graphics mode. This leaves only 5.75 Kbytes free for BASIC programs. Alternatively, `MODE 2` has the same amount of memory space available and a choice of 16 colours, yet the resolution is reduced to 160 by 256 pixels.

As the home microcomputer industry has developed, it is perhaps fair to say that more effort has gone into producing high-resolution graphics capabilities than anything else. Much of the increased memory of modern machines is dedicated solely for use by the graphics display. Whereas early machines like the ZX81 had a pixel resolution of only 64 by 48, many newer machines have resolutions 10 times greater.

REVERSE POLISH NOTATION

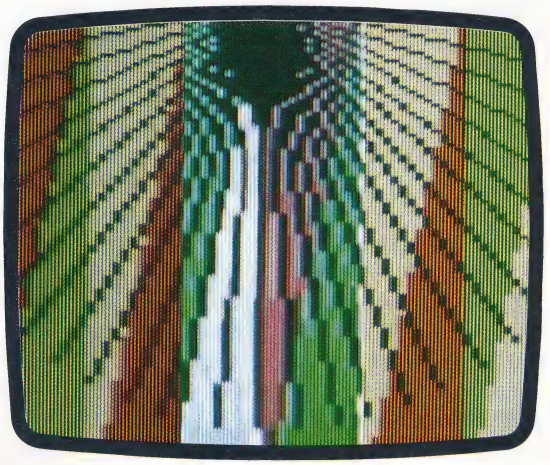
The normal method of writing arithmetical expressions is to place the operator between its operands. In Polish notation (see page 1340), the operator precedes the operands. However, in *reverse Polish notation*, the operands precede their arithmetical operators. For example, `x+y` in reverse Polish notation is written `xy+`. Similarly, more complex formulae, such as `x-y*z`, are described in reverse Polish notation as `xyz*-`. Note that in the second example not only are the operators placed after the operands, but also that the operators are themselves reversed. This is because it is the convention that, in the absence of brackets, multiplication and division are always evaluated before addition or subtraction. In the case where brackets are included to indicate the subtraction is to be evaluated first — for example, `(x-y)*z` — then the notation in reverse Polish would be `xy-z*`.

Polish notation is important because it reflects the way in which the computer operates. When a computer attempts to interpret a function it will store the numbers on the stack. As the stack is a LIFO (last in first out) device, it is essential that the numbers to be evaluated first are at the top of the stack rather than at the bottom, where they have to be retrieved before the evaluation of the expression can take place. Thus reverse Polish notation, which places the numbers in the reverse order to which they are to be evaluated, is ideally suited to microcomputer processes. Another advantage of using Polish notation in both its forms is that it makes the checking of legal functions much easier as all the expressions are grouped together.

RGB

This is a system used in microcomputers for producing high quality colour graphics on a VDU screen. Unlike composite video (see page 330) an RGB (Red, Green, Blue) signal is separated into its component colours by the modulator circuitry as they are received from the video chip. The three colours are then transmitted to the monitor along separate lines — one for each of the three video guns inside the cathode ray tube.

Because each of the signals is sent down a separate line, there is no interference between them and the resulting picture is much sharper.



IAN McKINNELL

Clear Advantage
Here we show the differences between the three most commonly used types of screen display. From left to right, these are a composite video display, a standard television picture and an RGB display. The RGB display is clearer than the other pictures as each of the three signals that make up the colour display are sent separately to the monitor

CRISPIN THOMAS

Popular Improvements
The Amstrad CPC 664 is an upgraded version of the very popular CPC 464 micro. Instead of a built-in cassette deck, this machine has a built-in Hitachi-standard 3in disk drive. The cursor keys have been expanded for ease of use, and in addition, Amstrad has improved the BASIC ROM to include ten additional commands as well as the DOS commands

PRESSING AN ADVANTAGE

As a follow-up to its very popular CPC 464, Amstrad has introduced the CPC 664, with a built-in disk drive and an all-in-one design like its predecessor. The available memory space, however, will need enlarging if the CPC 664 is to break into the small business market, as Amstrad hopes.

The Amstrad personal computer (CPC 464; see page 429) drew a considerable amount of praise when it was launched in 1984. Although the computer could not boast any technical breakthroughs, the machine proved a success at a time when home micro sales showed every sign of levelling off in the UK. A year later, the company launched a second machine, the Amstrad CPC 664 — essentially the same machine, but with a built-in disk drive instead of the cassette deck.

The layout of the CPC 664 is identical to that of the earlier CPC 464, with certain keys now in light blue rather than green. Amstrad has also decided to number the keys on the numeric keypad F1, F2 and so on, although the keys themselves perform exactly the same functions. The only other difference between the keyboards is the larger keys on the cursor cluster, which allow you to manipulate the cursor much more easily.

With a disk drive built in, the cassette deck keys have obviously disappeared and, in place of the cassette deck, there is now a Hitachi-standard 3in drive. The disk drive appears much smaller than the external disk drive (see page 1209) because the CPC 664 drive takes its power from a 12v supply on the monitor provided with the computer.

As the CPC 464 had its own built-in cassette deck, it had no need for a cassette port, but this facility has been added to the 664 so that users can take advantage of the software base that has built up on tape for the earlier machine.

Should users who own programs on cassette wish to transfer their material onto disk, Amstrad has made an arrangement with Timatic (a duplicating company) by which you can send them a cassette and the company will transfer it onto disk for the price of a blank disk.

The Amstrad CPC 464 has a single expansion bus that is intended not only for the addition of a floppy disk drive but also as a general purpose peripheral interface. The Amstrad 664 has not only a peripheral interface, but also an extra port for a second floppy disk drive. This is an important feature if the machine is to use CP/M, as many packages written to run under this operating system require that the applications disk should be in one drive while the data disk should be in another. Of course, you can get by with a single drive with CP/M, but it usually entails repeatedly having to swap disks in the drive.

In producing the new machine, Amstrad has taken the opportunity to upgrade the BASIC ROM to include some new commands. The disk operating system commands are there of course, and are identical to those fitted to the DDI ROM on the external floppy disk interface (see page 1209). Many of the new commands add features to the already powerful list of graphics commands available to the BASIC programmer. The most important of these is probably the FILL command, a curious omission from the original BASIC. This

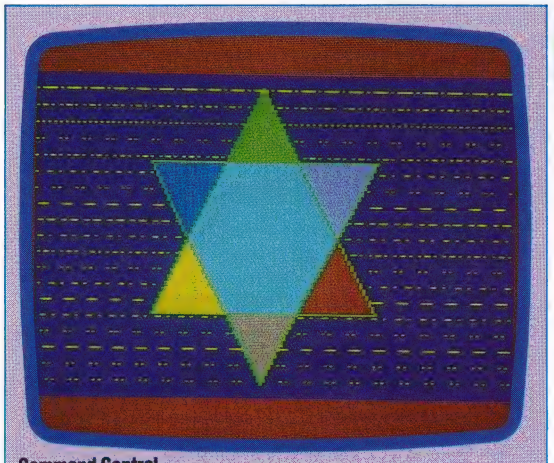
command will fill an area with either the current foreground colour or with a colour set by the programmer.

GRAPHICS COMMANDS

Also available to the graphics programmer is the MASK command, which can produce a dotted line. The command is written in the format MASK i, p where i stands for an integer between 1 and 255, and p determines whether the first point should be plotted or not. As its name suggests, the command is a 'mask', which performs a bitwise logical operation on the character cell. Thus MASK 1, p will produce a single dot every eight pixels, whereas MASK 17, p produces two dots. Setting MASK to 255 returns to a continuous line.

An associated pair of commands to MASK are GRAPHICS PEN and GRAPHICS PAPER. Normally when we draw a line we cannot see the background colour. However, when producing a dotted line with the MASK command, it is desirable to see the background. Thus GRAPHICS PAPER allows us to set the background graphics to either the current PAPER colour or some other colour. Similarly, GRAPHICS PEN sets the foreground colour for lines or points.

To refine further the process of writing graphics to the screen, an additional command, FRAME, has been added. Often, when graphics are produced, you will notice a certain amount of flicker on the screen. This is because the graphics are attempting to appear on the screen during the middle of a raster scan. The effect of the FRAME command is to halt the running of the BASIC program until the raster scan recommences at the top of the screen so that the graphics can be included in a single 'frame'. This command, therefore, produces a much smoother display.



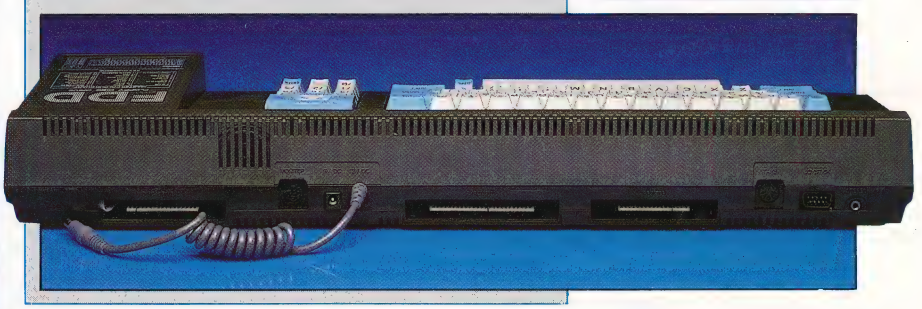
Command Control
This screenshot demonstrates three of the new commands that are available on the Amstrad CPC 664. The dotted lines in the background are drawn using the MASK command, while the points of the star are plotted using the DRAW command (which draws to a position relative to the current cursor position). Finally, the areas within the star are coloured using the FILL

To assist in the easy manipulation of data on the screen, COPY CHR$ has been added. This means that characters in one part of the screen can be transferred to another area. This is particularly useful for business applications where, for example, a list of figures or addresses in one window could be moved elsewhere for processing.

These are just a few examples of the number of commands that have been added to the BASIC. Although the ROM is intended to be compatible with the previous version of the BASIC, certain packages designed to run on the CPC 464 will not work on the 664. This is no fault of Amstrad's. Like many other manufacturers, the company has reserved the right to upgrade the BASIC when necessary. In order to preserve compatibility, the company included a jumpblock for vectoring calls to addresses in ROM. Unfortunately, in order to provide extra speed, some third party software manufacturers have bypassed the jumpblock and called the routines directly. Because many of these have now changed their address in the new version of the ROM, much of the software will no longer be valid.

The Amstrad CPC 664 is intended to be both a small business and a home hobbyist's computer. As such, the computer looks to be very good value. However, one or two question marks hang over the viability of the new machine as a business computer. To begin with, there is the perennial bugbear of a software base. Although Amstrad has adopted CP/M as a disk operating system, the disks that are used by its computers have so far failed to gain any widespread acceptance. In effect, this means that customers would normally have to wait for the software to appear in the correct format.

Reaching Out
Although the interfaces on the rear of the computer are similar to those on the CPC 464, some additions have been made. On the right, a cassette port has been added, while on the far left a floppy disk interface has been included, which frees the expansion bus for other uses. The lead attached to the computer draws power from the monitor to run the built-in disk drive



However, thanks to Amstrad's agreement with Timatic, 5¼in format CP/M disks can also be duplicated. There is also the possibility, if need be, of purchasing a suitable disk drive from a third party supplier to run in conjunction with the Hitachi unit.

Some of the standard CP/M packages, such as WordStar or dBase II, will not run on the Amstrad. This is because the computer has only 39 Kbytes free for applications programs when running under CP/M. However, a memory expansion board for the computer is promised that will enable users to take full advantage of the enormous CP/M software base.

AMSTRAD CPC 664

PRICE
With colour monitor £449; with green screen monitor £339. Both inc VAT

INTERFACES
Expansion bus, second floppy disk interface, cassette port, I/O port, joystick interface, printer port, 12v input, 5v input, monitor socket

SOFTWARE PROVIDED
The CPC 664 is bundled with CP/M 2.2 and Dr LOGO

DOCUMENTATION
The manual contains a full explanation of Amsoft BASIC, Dr LOGO and CP/M, and is a compilation of the CPC 464 and the DDI floppy disk manuals

STRENGTHS
At the price, it is hard to fault the CPC 664 as a small business machine. Only a few years ago a similar system would have cost several times as much

WEAKNESSES
Until it is rectified with the introduction of a memory expansion board, the computer will suffer from a lack of available memory space with which to run the most commonly used CP/M packages

SETTING THE SCENE

Designing adventure games—particularly those featuring interactive characters—demands a tightly structured approach to programming. Not only does this make them easier to debug but it also makes it possible for us to add new features or remove the less successful ones.

The program we will write to provide an environment for our interactive character routine will have the structure shown in our diagram. We'll work through the program step-by-step, starting with the initialisation routines. To keep things simple, we'll use string arrays to store all the data and we'll read the default values into the arrays from data statements.

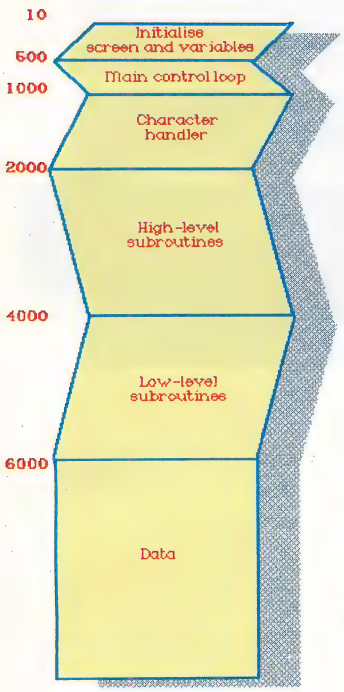
First and foremost, we require three location descriptions — one for each room in the infamous Dog and Bucket. We also need some means of knowing how the locations are connected—so that if a character moves east from the lounge, for example, we will know that he or she has ended up in the saloon. The two-dimensional array l$ holds all this information in the manner shown in our Array Table.

Deciding on how to represent the data for the objects in our game will depend primarily on what role they are expected to play. In this game, we are more concerned with objects as items for the characters to pick up, drop, or perhaps throw at each other. In the case of the cornish pasty and the ham sandwich, we will also want to consider the questions of edibility — and, of course, there should be plenty to drink! Object data structures, therefore, need to be able to cope with the following questions in our game: where is it, is it edible and is it drinkable.

To do this, we use the two-dimensional array b$ as shown in the Array Table. We will be able to check on the relevant elements of this array at any time during the program to answer any of the three questions just outlined.

Objects and locations pose few problems, so let's approach the more interesting task of deciding how to store information about our characters. The Dog and Bucket plays host to the following merry revellers; Toby Belcher, Fiona Frappe, Steve Swigg, Sally Short, Rupert Beer, and Molly Mixer. We will also include another character, Fred the barman. This latter character is not interactive — the barman is simply included in a location description and the character handler will occasionally display messages about his 'actions'. In this way, we can see that a character does not necessarily require complex programming and handling during run-time to be effective.

To decide what attributes we need to store for our characters, let's first recall the list of attributes we considered necessary for a computer-controlled 'person' (see page 1441). First, movement from one location to another is essential, so obviously we need to keep track of a character's position. Secondly, they must be able to manipulate objects, so a record of each character's inventory must be included somewhere along the line.



A Well-Made Play
Adopting a modular approach to programming will enable us to modify our game at a later stage if desired and make the adaptation of the character-handler to run with other programs less daunting. As in most adventure software, the bulk of memory is taken up by data, most of which will be text-messages for printing to the screen

Object Table

Object Number	Description	Start location	Edible?	Drinkable
1	a glass of beer	2	N	Y
2	an empty tin of catfood	3	N	N
3	a Dog and Bucket cornish pasty	1	Y	N
4	a bar-stool	2	N	N
5	an ashtray	1	N	N
6	a stale ham sandwich	2	Y	N
7	a pint of bitter	0	N	Y
8	a creme de menthe	0	N	Y
9	a whisky and water	0	N	Y
10	a neat vodka	2	N	Y
11	a pint of lager	0	N	Y
12	a gin and ginger ale	0	N	Y

Object Lesson
There are 12 different items to be found in the Dog and Bucket. Note that items 7 to 12 are 'owned' by the characters (see Character Table) and so, with the exception of item 10, start off at location 0 since they are being carried by their owners. At the start of the game, however, Sally Short has mislaid her drink (number 10), which is to be found in the saloon

INVOLVING YOURSELF

The other main attributes of a character concern communication with the player and awareness of the environment. The first does not, in fact, need to be implemented within the character-handler routine itself. To understand why, you only have to think about what happens

Character Table

	2 Location	3 Inventory	4 Strength	5 Mood	6 Object	7 Sex	8 LCH	9 LCD	10 Handle Frequency	11 Move Frequency
1 Tony Belcher	2	7	10	10	7	M	0	0	7	4
2 Fiona Frappe	1	8	30	10	8	F	0	0	3	5
3 Steve Swigg	1	9	8	10	9	M	0	0	4	6
4 Sally Short	2	0	20	10	10	F	0	0	5	5
5 Rupert Beer	2	11	10	6	11	M	0	0	4	6
6 Molly Mixer	1	12	15	6	12	F	0	0	5	5
7 You	1	0	255255	-	?	0	0	0	0	

DAVID HIGHAM

Dramatis Personae

Our table shows the initial values of the different attributes for each of the characters in the Dog and Bucket. All the characters start with their own drink in their inventory, with the exception of Sally Short.

Character number 7 represents the player and has been included to ensure that the other characters 'pay attention' to you! However, character 7's handle factor is zero, ensuring that your actions will remain your own responsibility and will not be subject to the manipulation of the character-handler routine

each time you enter a command in a game. If, for example, you type in Get dagger, and the dagger is present, the program will amend the variable that holds the dagger's position to indicate that it is now being carried, and then alter the player's inventory accordingly. If, on the other hand, you enter a command to another character to get an object, the program simply performs the same process but this time it alters the inventory of the character-concerned.

There is no need to involve the character handler routine at any point — the handler exists only to ensure that characters can lead their lives independently of the player. Of course, when the character handler is executed, it will find the dagger is now in the character's inventory and act accordingly.

The easiest way to ensure characters and the player get along together is perhaps the most obvious—to implement yourself as a character as well. All you have to do is define a character called you, and each time the character-handler routine is called, you will find yourself completely involved in the lives of your fictitious companions.

Communication with the player is not a difficult principle to grasp and implement effectively in the program (although such communication is always bound to be rather limited), but awareness of the environment — the final prerequisite for an interactive character — can be very tricky to include. To a limited extent, this feature can be included in our character-handler simply on the basis of the data we have already listed. The routine can check to see whether objects are present, manipulate them or even generate some intelligent remark about them to be 'spoken' by a character. It can also check on a character's location and perhaps cause the character to make a comment about it. However, although useful, routines like this are rather limited. What is obviously required is an awareness of other characters and, in particular, the ability to give a computer-controlled 'person' some kind of 'continuity'— in other words, to give characters a sense of history, as we discussed earlier in the series (see page 1441).

To do this, we'll introduce two new attributes to be stored for each character — the last command code (LCD) and the last character code (LCH). These indicate respectively the last action taken by (or, in some cases, inflicted upon) the character, and the other party (if any) involved in this action. These can be included in our character data array and as an example of how they might be used, consider the case in which Toby Belcher (character number 1) gives a pasty to Fiona Frappe (character number 2). Fiona's LCD will now be set to indicate a 'receive' action, and her LCH will hold the number 1. The next time the handler routine is called, it will check the data and be able to deduce from the object held in Fiona's inventory (the pasty) what has just happened. The routine can then decide whether or not Fiona should say Thank you; if she does, her LCD will be set to indicate 'communicate' and her LCH will still hold 1.

If, on the other hand, Fiona does not take any action following Toby's donation, her LCD and LCH will both be set to 0, indicating that the past has, effectively, been forgotton. The use of LCD and LCH can in fact become quite complex, as we'll see later in the series when we examine it in greater detail. In the meantime, the codes we shall be using for LCD are included in the Array Table.

There are six other attributes that we will be storing for our characters, each of which is useful and very straightforward:

- *Strength.* This attribute determines the character's ability to move and communicate. In this particular case, a character will be treated as unconscious if its strength drops to or below zero. Certain actions (such as drinking too much) will

Array Table

Array	Dimensioned To	Holds Information Concerning
I$	3,5 (Spectrum—3,5,255)	location description (I$(n,1)) and the four exit codes (I$(n,2...5))
b$	12,4 (Spectrum—12,4,30)	object description (b$(n,1)) and location/edible/drinkable (b$(n,2,...4))
c$	7,11 (Spectrum—7,11,15)	character names (c$(n,1)) and their attributes (c$(n,2,...11))

Last Command Codes

LCD	Indicates
1	The last time the character-handler was called, the character received the object in its inventory from the character indicated by LCH
2	The character previously said something to the character indicated by LCH
3	Monologue. The character has just made a remark. This code is used to stop characters from continuously making the same, or similar, remarks
4	Get. The object in the character's inventory was picked up during the last execution of the character-handler
5	Hit. The character has just been hit by an object thrown by another character
6	Eat. The character is in the process of eating the object in its inventory
7	Enter. The character has just entered the current location

Holding Pattern

The three principal arrays (I$,b$, and c$) hold all the necessary data concerning locations, objects and characters. Note that zero elements are not used (to aid compatibility with different BASICs, some of which do not have a zero element facility). String arrays will be used to hold both alphanumeric and numeric data (using STR$ and VAL$). The last command codes (LCDs) are used by the character handler to give some small degree of continuity, or 'history', to a character and are held in c$(n,9) — see Character Table.

Location Table

Location	Description	N	S	E	W
1	You are in the lounge of the Dog and Bucket. Several shady characters are gathered together in a corner playing dominoes. Behind the counter, Fred the barman looks his usual cheery self. A door leads east.	0	0	2	0
2	Behold the saloon of the Dog and Bucket, which looks as if it could do with extensive redecoration. The floor appears to have been regularly hosed down with beer slops. Doors lead west and south.	0	3	0	1
3	Ugh! This is the kitchen, where the famous Dog and Bucket cornish pasties are prepared for an ever-hungry clientele. You notice a number of empty cat-food tins, which is strange because there are no cats.	2	0	0	0

Exit, Stage West

The three locations each have their own description together with four exit codes, one for each point of the compass. The exit code simply holds the number of the destination; so, for example, moving north from the kitchen leads to location number two, the saloon. A location number of zero indicates that movement in that direction is not possible

reduce a character's strength—others might increase it.

● *Mood.* This determines the way in which characters treat each other. For example, if the handler routine finds that a character is carrying a particular object, it may reach a point where it has to decide whether the object needs to be dropped or retained. If the character's mood is particularly poor (say, near zero), the handler may dictate that the object should be thrown at someone. Factors that influence mood in the Dog and Bucket include losing one's drink.

● *Object.* Each character in the program has his or her 'own' object, the number of which is recorded by this attribute. In this particular case, the objects are all drinks of various descriptions, and losing or finding one's drink has certain effects on a character's behaviour.

● *Sex.* This attribute records whether a character is male or female. It is used, for example, to determine the correct pronouns to be used in messages.

● *Handle Frequency.* The handle frequency (HF) indicates the amount of time that must elapse before the character is checked by the character-handler routine. For example, a character with an HF of 5 would be operated upon by it every fifth time the routine was called. An HF of 0, however, would mean that the character was not to be updated by the handler routine at all. Together with the move frequency attribute (following) the HF can be used to 'freeze' (setting HF to 0), slow down the characters (increasing HF), or to give them the appearance of being more active (decreasing HF).

● *Move Frequency.* We might wish certain characters to be active with regard to their companions and surroundings, but need at the same time to keep them in one location or ensure that they do not move about too quickly. The move frequency determines how often the handler routine will attempt to move the character.

Our chart shows the initial values for each of the character attributes.

MAKING A MOVE

In the previous instalment, we developed a routine for our game of Go that enables the computer to select moves to defend its own groups or attack others. Here, we design a routine for the computer to play sensible strategic moves in the absence of defensive or attacking situations.

The game of Go is probably one of the most difficult board games to try to computerise. The sheer size of the board and flexibility of the moves severely restrict the use of look-ahead game trees, which are commonly found in the majority of computerised games of skill, including chess, backgammon and draughts.

In a game such as chess, where there is a fairly limited number of moves available, averaging around 30 from any given position, it is feasible to examine them all in detail, thus ensuring that the computer will find at least one move to play, even if the move does consist of resigning by turning the king on its side! Unfortunately, our evaluation system for Go cannot effectively work in this manner. Our group evaluation routine will only check for moves next to groups with less than three liberties. If there are no board groups in this situation, the routine will not find any moves. In future instalments we will devise other evaluation routines based around simple pattern-matching techniques, which will also only find moves in certain situations. What we really need is a 'catch-all' evaluation, which is always sure of finding at least one possible move, assuming that a move is available.

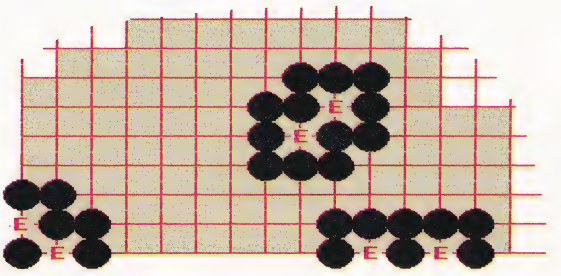
The easiest way of doing this would be just to play any available move at random. This routine would look something like:

```
5000 DEF PROCrandom_move
5010 LOCAL L%
5020 :
5030 FOR L%=17 TO 255
5040 IF FNlegality (L%, black%) = 0 THEN
       location% = L%
5050 NEXT
5060 ENDPROC
```

This routine is bound to return a legal move if one exists. The routine could be improved by starting the loop at a random board position, and possibly assigning a score based on the number of liberties available to the placed stone (L%) — this will be automatically calculated and returned from FNlegality in the variable clib%. However, even with these improvements, the computer is not going to play very sensible moves, unless by luck!

What is really needed is a routine which will play random moves, but only to fairly sensible positions. One way of achieving this objective is to 'weight' the board. This is a technique frequently used in games such as Othello, where certain board positions are more advantageous than others.

Although it is difficult to decide the advantages and disadvantages of specific positions, certain areas of the board are considered better than others. For example, consider the number of pieces necessary to completely surround two separate liberties. You'll remember that these are called 'eyes', and any group with two or more eyes is safe from capture. The number of stones needed to form a safe group in this way varies, depending on whether the group is in the corner, the side, or the middle of the board. Only six stones are required to form a save group in the corner, and this is obviously better than the twelve stones needed in the middle of the board.



The most likely first moves will be on or near the corner handicap positions. From here, in the usual course of play, players extend along the sides of the board (on the third or fourth lines) in an attempt to gain territory. Then, if attacked, you have the advantage of being able to use your fourth-rank corner stone as a base from which to get two eyes in the corner.

Having explained these basic tactics, the board weightings given in the diagram should make

some sense. These are symmetrical about all four corners of the board, being set up by the routine between lines 1020 to 1230. (Note that the routine is given out of sequence in the Commodore 64 version, between lines 363 and 383, due to the lack of a RESTORE line-number command to reset the data pointer.)

The numbers assigned to the various positions have been chosen based on the tactics just described, but are by no means the best. You might like to try different weightings to check their effect. One way of 'tuning' these weightings is to have the computer play itself, with each side using a different weighting table. This can be done in the same way as we set up the computer to referee a two-player game. Just change lines 60 and 80 to read:

```
60 move% = move% + 1: black% = 2: white% = 1:
   PROCblack_move
80 move% = move% + 1: black% = 1: white% = 2:
   PROCblack_move
```

```
220 DIM board% 255, weight1% 255, weight2% 255
```

The PROCread_weights routine can then be changed to place one set of weightings in the weight1% byte array, and the other after weight2%. Now by adding weight% = weight1% to line 60, and weight% = weight2% to line 80, the computer will use different board weightings for each player.

Having set up the weighted board, we can add the PROCfind_any_move 'catch-all' routine. This is similar to the random-move routine, but uses the board weightings to find a 'reasonable' move. The

Module Five

BBC Micro:

```
220 DIM board% 255,weight% 255
1010 :
1020 DEF PROCread_weights
1030 LOCAL A%,B%,C%,D%,X%,Y%,V%
1040 RESTORE 1150
1050 FOR Y%=1 TO 8
1060   FOR X%=Y% TO 8
1070     A%=16*Y%+X% : C%=16*Y%+(16-X%)
1080     B%=16*X%+Y% : D%=16*X%+(16-Y%)
1090     READ V%
1100     weight%?A%=V%:weight%?(272-A%)
=V%
1110     weight%?B%=V%:weight%?(272-B%)
=V%
1120     weight%?C%=V%:weight%?(272-C%)
=V%
1130     weight%?D%=V%:weight%?(272-D%)
=V%
1140   NEXT:NEXT
1150 DATA 0,1,1,2,2,2,2,2
1160 DATA   4,5,5,4,4,3,3
1170 DATA     7,7,6,6,5,5
1180 DATA       7,6,5,5,4
1190 DATA         6,5,4,3
1200 DATA           4,3,2
1210 DATA             2,1
1220 DATA               0
1230 ENDPROC
1240 :
1250 REM**************************
1350 PROCread_weights
2620 IF location%=0 THEN PROCfind_any_m
ove:T$="RND"
2630 IF location%=0 THEN end%=TRUE:ENDP
ROC
2820 score=(8*S%/L%-clib%+2*L%)*weight%
?tloc%(Q%)
3510 :
3520 DEF PROCfind_any_move
3530 LOCAL L%,hi,score
3540 hi=-9999
3550 FOR L%=17 TO 255 :score=RND(1)+wei
ght%?L%
3560   IF (L% AND 240)=0 OR (L% AND 15)
=0 OR score<=hi THEN 3580
3570   IF FNlegality(L%,black%)=0 AND c
lib%>2 THEN hi=score:location%=L%
3580   NEXT
3590 ENDPROC
3600 :
3610 REM**************************
3800 board%?P%=0:IF C%=black% THEN weig
ht%?P%=0
```

Commodore 64:

```
220 BOARD=49152:WEIGHT=BOARD+256
305 GOSUB 363
362 :
363 REM READ-WEIGHTS ROUTINE
364 RESTORE:FOR X=1 TO 4:READ Y:NEXT
365 FOR Y=1 TO 8
366 FOR X=Y TO 8
367 A%=16*Y+X:C%=16*Y+(16-X)
368 B%=16*X+Y:D%=16*X+(16-Y)
369 READ V%
370 POKE WEIGHT+A%,V%:POKE WEIGHT+272-A%
,V%
371 POKE WEIGHT+B%,V%:POKE WEIGHT+272-B%
,V%
372 POKE WEIGHT+C%,V%:POKE WEIGHT+272-C%
,V%
373 POKE WEIGHT+D%,V%:POKE WEIGHT+272-D%
,V%
374 NEXT:NEXT
375 DATA 0,1,1,2,2,2,2,2
376 DATA   4,5,5,4,4,3,3
377 DATA     7,7,6,6,5,5
378 DATA       7,6,5,5,4
379 DATA         6,5,4,3
380 DATA           4,3,2
381 DATA             2,1
382 DATA               0
383 RETURN
384 :
385 REM**************************
1350 GOSUB 363
2620 IF LOCAT%=0 THEN GOSUB 3520:T$="RND
"
2630 IF LOCAT%=0 THEN FIN%=-1:RETURN
2820 SCR=(8*BS/BL-CLIB%+2*BL)*PEEK(WEIGH
T+TLOC%(Q))
3510 :
3520 REM FIND-ANY-MOVE ROUTINE
3540 HI=-9999
3550 FOR I=17 TO 255:SCR=RND(0)+PEEK(WEI
GHT+I)
3560 IF (I AND 240)=0 OR (I AND 15)=0 OR
 SCR<=HI GOTO 3580
3570 LP%=I:LC%=BLACK%:GOSUB 3890:IF LL%=
0 AND CLIB%>2 THEN HI=SCR:LOCAT%=I
3580 NEXT
3590 RETURN
3600 :
3610 REM**************************
3800 POKE BOARD+RP%,0:IF RC%=BLACK% THEN
 POKE WEIGHT+RP%,0
3805 SK%(STACK%)=RP%:STACK%=STACK%+1
```

(See the Basic Flavours box for the Amstrad, Commodore 64 and Spectrum equivalents.) To add the different weightings, first change line 220 to read:

variable clib% is also checked to ensure that the computer doesn't play into a position which would leave one of its groups with less than three liberties. This routine is called from line 2620, with the RND comment. This shows which routine was used to find the last computer move, in the same way as the GP comment in the previous group evaluation

routine. If this new routine cannot find a possible move, then line 2630 sets the end of game variable and exits from the program.

Since we have gone to the trouble of setting up board weightings, we can also use them to improve other evaluation routines. This can be done in the group evaluation routine by multiplying the score by the board weighting (line 2820). Again, you might like to try changing this score in an attempt to improve the computer's performance.

One evident problem was the tendency for the program to play stones in areas of previously captured stones. Based on our evaluations, obviously the first stones played are going to be placed into what the program considers to be the better board positions. If you capture some of these stones, then these previously 'good' positions again become vacant. The computer, noticing this, immediately starts playing stones back into these areas which are obviously already surrounded by your stones — not a good idea! Line 3800 has therefore been added to give a simple form of 'dynamic' board weighting. This just means that the board weightings change as the game progresses. For instance, in the game of chess, you might weight the board for the king. During the first few moves of the game you would want high weightings in the corner to encourage the king to take up a safe position. Whereas, when you reach the end game, you will probably want to change the weightings to tempt the king into a more commanding central position.

In our Go program, line 3800 is part of the PROCremove routine, which removes captured stones from the board. If these stones are black, this line will also change the board weighting for that position to zero. From then on, although black may still play into this position, it is far less likely.

Amstrad CPC 464/664:

```
220 board=&A000:weight=&A100
1010 :
1020 REM read weights routine
1040 RESTORE 1150
1050 FOR y%=1 TO 8
1060 FOR x%=y% TO 8
1070 a%=16*x%:c%=16*y%+(16+x%)
1080 b%=16*x%+y%:d%=16*x%+(16-x%)
1090 READ v%
1100 POKE(weight+a%),v%:POKE (weight+272
-a%),v%
1110 POKE(weight+b%),v%:POKE (weight+272
-b%),v%
1120 POKE(weight+c%),v%:POKE (weight+272
-c%),v%
1130 POKE(weight+d%),v%:POKE (weight+272
-d%),v%
1140 NEXT x%,y%
1150 DATA 0,1,1,2,2,2,2,2
1160 DATA 4,5,5,4,4,3,3
1170 DATA 7,7,6,6,5,5
1180 DATA 7,6,5,5,4
1190 DATA 6,5,4,3
1200 DATA 4,3,2
1210 DATA 2,1
1220 DATA 0
1230 RETURN
1240 :
1250 REM **********
1350 GOSUB 1020:REM read weights
2620 IF location%=0 THEN GOSUB 3520:T$="
RND":REM any move
2630 IF location%=0 THEN over%=1:RETURN
2820 score=(8*gs%/gl%-clib%+2*g1%)*PEEK(
weight+tloc%(q%))
3510 :
3520 REM find any move routine
3540 hi=-9999
3550 FOR i%=17 TO 255:score=RND(1)+PEEK(
weight+i%)
3560 IF (i% AND 240)=0 OR (i% AND 15)=0
OR score<=hi THEN 3580
3570 lp%=i%:lc%=black%:GOSUB 3890:IF 11%
=0 AND clib%>2 THEN hi=score:location%=i
%
3580 NEXT i%
3590 RETURN
3600 :
3610 REM **********
3800 POKE (board+rp%),0:IF rc%=black% TH
EN POKE (weight+rp%),0
```

Sinclair Spectrum:

```
220 LET board=64000:LET weight
=board+256
1020 REM read weights routine
1040 RESTORE 1150
1050 FOR y=1 TO 8
1060 FOR x=y TO 8
1070 LET a=16*y+x: LET c=16*y+(1
6-x)
1080 LET b=16*x+y: LET d=16*x+(1
6-y)
1090 READ v
1100 POKE weight+a,v: POKE weigh
t+272+a,v
1110 POKE weight+b,v: POKE weigh
t+272-b,v
1120 POKE weight+c,v: POKE weigh
t+272-c,v
1130 POKE weight+d,v: POKE weigh
t+272-d,v
1140 NEXT x: NEXT y
1150 DATA 0,1,1,2,2,2,2,2
1160 DATA 4,5,5,4,4,3,3
1170 DATA 7,7,6,6,5,5
1180 DATA 7,6,5,5,4
1190 DATA 6,5,4,3
1200 DATA 4,3,2
1210 DATA 2,1
1220 DATA 0
1230 RETURN
1250 REM **********
1350 GO SUB 1350:REM read weights
2620 IF location=0 THEN GO SUB
3520: LET t$="RND"
2630 IF location=0 THEN LET end
=1: RETURN
2820 LET score=(8*gs/gl-clib+2*g
l)*PEEK (weight+j(q))
3510:
3520 REM find-any-move routine
3540 LET hi=-9999
3550 FOR i=17 TO 255: LET score=
RND+PEEK (weight+i)
3560 IF INT (i/16)=0 OR i-16*INT
(i/16)=0 OR score<=hi THEN GO
TO 3580
3570 LET lp=i: LET lc=black: GO
SUB 3890: IF 11=0 AND clib>2 THE
N LET hi=score: LET location=i
3580 NEXT i
3590 RETURN
3610 REM **********
3800 POKE board+rp,0: IF rc=blac
k THEN POKE weight+rp,0
```

Basic Flavours

The program can be set up to allow the computer to play itself by making these alterations. By setting up two different weighting tables the computer will select each table alternately, enabling the weighting tables to be compared

Amstrad CPC 464/664:

```
60 mve% = mve% + 1: black% = 2:
   white% = 1: weight = weight1:
   GOSUB 2540
80 mve% = mve% + 1: black% = 1:
   white% = 2: weight = weight2:
   GOSUB 2540
220 board = &A000: weight1 = &A100:
    weight2 = &A200
```

Spectrum:

```
60 LET move = move + 1: LET black = 2:
   LET white = 1: LET weight = weight1:
   GO SUB 2540
80 LET move= move + 1: LET black = 2:
   LET white = 1: LET weight = weight2:
   GO SUB 2540
220 LET board = 64000: LET weight1 =
    board + 256: LET weight2 = board +
    512
```

Commodore 64:

```
60 MOVE% = MOVE% + 1: BLACK% = 2:
   WHITE% = 1: WEIGHT = W1: GOSUB
   2540
80 MOVE% = MOVE% + 1: BLACK% = 1:
   WHITE% = 2: WEIGHT = W2: GOSUB
   2540
220 BOARD = 49152: W1 = BOARD +
    256: W2 = BOARD + 512
```

BREAKER, BREAKER

Adding your own BASIC commands to those of the Sinclair Spectrum can be achieved using the Interface 1 ROM. In the previous instalment, we looked at the 'theory' behind the addition of new commands; now we show you how to go about actually doing it.

Any new command that you want to add to the Spectrum BASIC must first fail the syntax and run-time checks (see page 1478) made by both the main ROM and the Interface 1 ROM. In this case, control will, eventually, pass to the routine with its address held in the VECTOR system variable at 23735 and 23736. There are two ways of doing this.

1. Modify a current keyword so that it will fail these checks. This can be done by passing an incorrect number of parameters, by using it out of its normal syntax or adding a character — known as a *breaker* — to generate an error. For example:

BORDER * (adding a breaker after a command)
LINE 10, 10, 1, 19 (too many parameters specified)

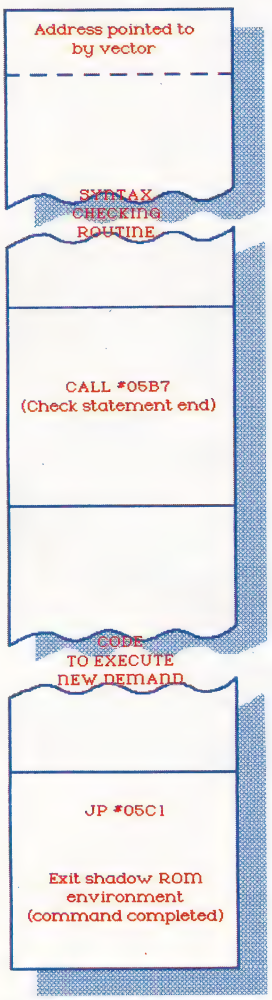
2. Add a totally new command by prefixing a command word with a breaker — such as *BEEP or *PRINT.

These commands will, of course, have access to any BASIC variables that you like. We will use the second method given here to add our commands.

Let's say that we wish to add a command, called *B, which generates a one-second tone. The first thing we need to do is decide where in memory this new code is to go. This should not be in a REM statement, since the insertion of the Interface 1 system variables makes the location of data within the program area unpredictable. We use, instead, CLEAR nn to produce some space.

For any new command that we wish to add, we need to structure our programs to include the steps shown in the diagram. When the routine pointed to by VECTOR is entered after the BASIC interpreter has encountered an error, we need to get hold of the character in the text of the BASIC program that has caused the problem to see whether it is, in fact, one of our own additional commands. This character is pointed to by the system variable CHADD (located at 23755 and 23756).

In order to retrieve the character pointed to by CHADD, we need to make use of two routines in the main ROM — GETCHAR and NEXTCHAR. The operations of these routines are described in detail in the Useful Addresses box, and either can be called while in the shadow ROM environment by using RST #10 followed by the address (either #0018 or#0020) of the routine required.


Address pointed to by vector
SYNTAX CHECKING ROUTINE
CALL #05B7 (Check statement end)
CODE TO EXECUTE NEW DEMAND
JP #05C1
Exit shadow ROM environment (command completed)

Interface Interceptor

The Spectrum responds to an error with an RST #08. As soon as the Interface 1 hardware detects that the Z80 program counter holds this address, it pages in the Interface 1 ROM and a jump is made to the routine whose address is held in VECTOR. This enables us to add new commands to the Spectrum, using a program structure as shown here

On entry to the routine pointed to by VECTOR, CHADD will hold the address of the character that has generated the error. Therefore, to execute our simple *B command we need to assemble the following listing:

```
D7      vector: rst  #10
1800            defw #0018              ;address of GETCHAR
FE2A            cp   "*"                ;is it a "*"?
C2F001          jp   nz,#01F0           ;if not, error
D7              rst  #10
2000            defw #0020              ;address of NEXTCHAR
FE42            cp   "B"                ;is it a "B"
C2F001          jp   nz,#01F0           ;if not, error
D7              rst  #10                ;get NEXTCHAR, should...
2000            defw #0020              ; ...be end of statement
CDB705          call #05B7              ;check end of statement
        runtime:
```

The shadow ROM routine at #05B7 must be entered with the A register holding the relevant character, pointed to by CHADD. You can, of course, achieve the same result using NEXTCHAR, as above. At syntax checking time, the routine at #05B7 causes a return to the main ROM to check the next statement, but at run-time it will be returned from like a normal subroutine, and the code at the label RUNTIME will be executed.

Note the ease with which we check each character of the command name. You could easily make the routine accept lower and upper case characters as part of the command name. In each check, should the letter not be the one expected, we must exit via #01F0 in the shadow ROM (version 1) so as to signal a syntax error. Note also that when we first enter the routine pointed to by VECTOR, we need to get the *current* character pointed to by CHADD, so we use GETCHAR at #0018,rather than NEXTCHAR at #0020.

The run-time code for our simple routine is very straightforward and simply produces a tone using the main ROM 'beep' routine, as shown below:

```
11058I runtim: ld   de,#0105           ;duration of 1 second
210007         ld   hl,#0700           ;pitch
D7             rst  #10
B503           defw #03B5              ;execute BEEP routine
C3C105         jp   #05C1              ;exit
```

The jump to #05C1 finishes things off cleanly, passing control back to the main ROM.

You could, of course, add a new BASIC command that makes use of any of the other main ROM routines that we have covered in the course (and the graphics ROM routines which we will be looking at in the next and final instalment) by calling them using RST #10. As an example of this, the following listing for the *B command uses the routines at #1C82 and #1E94 (both of which are detailed in the Useful Addresses box) to evaluate a single numeric parameter supplied by the user. The format of the command is *B n, where n is any number between 0 and 255. The parameter determines the length of the BEEP in seconds, so *B 255 will generate a 255-second tone.

```
              org  60000
      VECTOR: equ  #5CB7                ;VECTOR address
CF    init:   rst  #8
31              defb 49                 ;set up IF1 variables
2169EA          ld   hl,newcom          ;add of new command
```

```
22B75C          ld   (VECTOR),hl          ;get it in VECTOR
C9              ret                       ;end of initialisation
                ;
D7      newcom: rst  #10                  ;VECTOR jumps here
1800            defw #0018                ;GETCHAR
FE2A            cp   "*"
C2F001          jp   nz,#01F0
D7              rst  #10
2000            defw #0020                ;NEXTCHAR
FE42            cp   "B"
C2F001          jp   nz,#01F0
D7              rst  #10
2000            defw #0020
D7              rst  #10
821C            defw #1C82                ;evaluate parameter
CDB705          call #05B7                ;check statement end
D7      runtim: rst  #10
941E            defw #1E94                ;get param in a
47              ld   b,a
C5      loop:   push bc                   ;save counter
110501          ld   de,#0105             ;duration
210007          ld   hl,#0700             ;pitch
D7              rst  #10
B503            defw #03B5                ;call BEEP routine
C1              pop  bc                   ;restore counter
10F3            djnz loop
C3C105          jp   #05C1                ;exit - all done
```

Note that because the routine at #1C82 will only accept an eight-bit value (because it stores this value in the A register), so entering values outside the range 0 to 255 will generate an error. One other thing to notice is that during the generation of the tones, the FRAMES counter will not be incremented and the keyboard will not be scanned, so don't enter 255 as a parameter unless you're prepared to wait around a bit!

EXTRA PARAMETERS

So far we've concentrated on the method of adding commands by entering non-standard inputs to generate an error. You can, of course, use the Sinclair BASIC keywords as well, though they will have to be modified in order to generate the required error. The easiest way to do this is generally to add an extra parameter, so the following would all cause a jump to whatever routine was pointed to by VECTOR:

```
CIRCLE x,y,z,n,
LINE x
BORDER x,y,z
```

If you adopt this approach, CHADD will point to the offending keyword token on entry to the routine pointed to by VECTOR. This can be checked against the appropriate token code, which can be found in the Spectrum manual.

Despite the apparent complexity of the process, adding new keywords to Spectrum BASIC is in fact quite simple. The important thing is to remember which ROM is paged in at any one time. Should you exit the shadow ROM environment, you can always use hook code 50, plus the appropriate address, to call a shadow ROM routine.

In the final instalment of our series on the Spectrum operating system, we will take a look at the layout of the screen, so that you can conclude your exploration of the ROMs with a flourish of machine code graphics.

Interface 1 Identikit

If you intend to access shadow ROM routines directly, it is important to check which version of the Interface 1 ROM you are using. As a general rule, IF1's with serial numbers greater than 87315 have version 2 of the ROM, but a more reliable way of checking this is to enter:

```
CLOSE #0:PRINT PEEK 23729
```

This will return 0 for version 1 and 80 for version 2. Fortunately, the IF1 ROM calls that we've used to extend the BASIC are at the same addresses in both versions of the ROM. However, you may encounter problems using other routines. If in doubt, and you have a suitable disassembler, you might like to try checking the routines you are calling using our shadow ROM Loader program published in the last instalment (see page 1479).

Other points to note include the addition of two new hook codes in the version 2 ROM. Hook code #33 retrieves a Microdrive file record descriptor, and hook code #34 opens an RS232 'b' channel. In the latter case, the base address of the channel is returned in DE

Useful Addresses

In The Shadow ROM:

#01F0	Exit shadow ROM via main ROM error routine
#05B7	Returns during run-time if character held in A register is #0D (ENTER) or a colon. Otherwise generates an error
#05C1	Returns to main interpreter when a new command has been accepted

In The Main ROM:

#0018	GETCHAR — Gets CHADD (see below) in HL; gets character pointed to by CHADD in A; checks to see if character is printable and returns if it is. Otherwise passes on to NEXTCHAR
#0020	NEXTCHAR — Increments CHADD and places it in HL. Gets character pointed to by CHADD and checks to see if it is printable. If it is then NEXTCHAR returns with character in A. Otherwise the process is repeated
#1C82	Evaluates or checks a numeric expression, the first element of which must be pointed to by CHADD. On exit, CHADD points to the first character after the expression, and the result is placed on top of the main ROM calculator stack
#1E94	Takes value from top of the main ROM calculator stack and returns it in A if in the range 0 to 255. Otherwise generates an error

The system variable CHADD points to the current character being interpreted, and is located at addresses 237 5 and 23756

OVER THE MOON

Some computer games demand more than just lightning reflexes and a steady hand. Strategy games, like the enduringly successful Football Manager from Addictive Games Limited, require you to juggle a number of factors and make careful decisions to achieve the desired result.

Launched in the early 1980s, Football Manager has generated consistent long-term sales. In a market where most packages have only a limited shelf-life, no matter how successful they prove on release, the game stands out from its competitors. Yet there is nothing spectacularly innovative in it; the concept behind the game is very simple. You are placed in charge of a team in the Football League and must steer it to success in the FA Cup.

The game starts, naturally, at the beginning of a new season, with the team you are to manage in Division Four. You are then presented with a menu of such options as checking your position in the League Table, looking at details of the fixtures list, and considering the attributes of the players available for the squad.

The players are divided into three categories: defenders, midfield and attack. Obviously, a sensible manager will field a balanced team in every game. Before a match you are given the option of making alterations to your line-up in order to field the best possible side. However, in the early stages of the game you have only 12 players available, and so there isn't much to choose from. However, as the game progresses, additional players come onto the transfer market and you can 'bid' for them.

A player has three basic features, which should be borne in mind when bidding. First of all, they each have a 'skill factor', measured on a scale of one to five. Secondly, they each have a 'value', which although related to their skill factor (5,000 for every skill point) can vary within the transfer market. Finally, there is the player's 'energy'. This is on a scale of one to 20 and is decremented every time a player is fielded in a match. Thus, it is not advisable to field all your best players in every match, but rather to rotate them so that they can get periodic rests to rebuild their energy to a maximum. Of course, this is very difficult with a squad of only twelve players, and faced with a diminishing energy factor for the players, even the most careful manager will be forced into the transfer market sooner or later.

At the beginning of each 'round' a player will be offered for sale, and you are invited to make a bid for him. If your bid is accepted the player joins the squad. However, often the bid will be refused, even though you may be offering more than his stated value. You will then be asked to make another bid for the player, only this time his value will be increased due to your interest in him.

Once you have completed your forays into the transfer market, arranged loans where necessary, and examined your current league position, you then have the option to play a match. The computer displays the relative strengths of your team and that of the opposition, and you have the option of changing the team, removing tired and injured players and replacing them with fresh ones. Once this has been completed, you are ready to play. The computer will then present 'edited highlights' of the game.

EDITED HIGHLIGHTS

The highlights simply consist of goalmouth scrambles at either end of the pitch — the amount of time spent at one end of the pitch or the other is dependent on the relative strengths of the midfield, while the strengths of the opposing attacks and defences govern whether goals are scored. Although the match graphics add a great deal of excitement to the game as you watch the ebb and flow of the play, it should be said that the graphics display is very primitive — the BBC version, for example, is simply made up of block graphics characters.

At the end of the match the final score is displayed, together with the other results in the Division, and your team will move up or down the table accordingly.

The game requires that the manager not only consider the best strategy for the next match but must also look ahead to forthcoming matches, such as games with top of the table teams and cup matches where the best team is required. Therefore, decisions need to be made whether to rest your best players now (and risk dropping points) or to play them (and risk injury or a drop in energy, which could severely affect their performance in the important match).

Although the game contains no 'arcade' sequences as such, it is easy to see why it has remained so popular. Despite some limitations, it is a realistic enough simulation to make the player believe he or she really is managing a team, and when your team finally wins the First Division championship, there is a real sense of achievement.

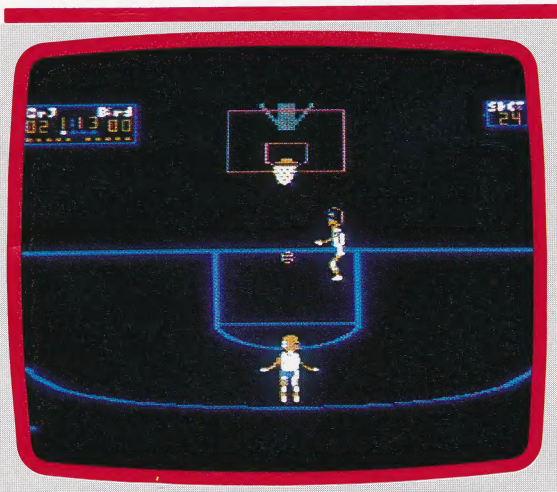
LIZ HEANEY

The Waiting Game
This screen, from the Commodore 64 version of Football Manager, shows the 'action sequence' from the game. At this point, the player has no control over the events taking place on the field but watches the 'edited highlights'. This is perhaps the most nerve-racking part of the game, as you wait to see whether your judgement is well balanced

Football Manager: For the ZX81, Sinclair Spectrum, Commodore 64, Vic-20, BBC Micro, Electron, Amstrad and Dragon
Price: Vic-20, Dragon, ZX81: £5.95; Sinclair Spectrum: £6.95; all others: £7.95
Publishers: Addictive Games Ltd, 7A Richmond Hill, Bournemouth, Dorset BH2 6HE
Author: Kevin Toms
Format: Cassette
Joysticks: Not required

It is not surprising that many successful games packages in the US are based on popular American sports. Yet many of the computer versions of these games also sell remarkably well in the UK, where the real sports aren't so popular. Jon Kaye gets the measure of two of the best sellers.

SPORTS USA



In the US, basketball is enormously popular, although the two-team version is not the only one played. The game of one-on-one, which pits one player against another using just half of the court, is an alternative to a full game, especially when 10 players can't be found. It is also a great exercise in basketball skills. Ariolasoft's One-On-One has attempted to capture the spirit of the game by producing a package that's been endorsed by two of the most popular figures in US basketball — Julius Erving (a.k.a. Dr J) and Larry Bird.

The actions and movements of the players on screen, though not spectacularly drawn, are nevertheless very accurate, right down to the reverse lay-up shot 'Big Bird' is so famous for. All of the rules of the game have been incorporated and each player's style has supposedly been emulated, though this could only be discerned by the most diehard of fans.

There are four levels of play, and several different modes to choose from — depending on the variations in the rules, one or two-player games and which personality is controlled by the players or computer. The 'shot clock', allowing you just 24 seconds to shoot, and the 'fatigue bar' must be kept in mind at all times, as must the general rules of play, especially at the university and professional levels where the referee is very pernickety about violations.

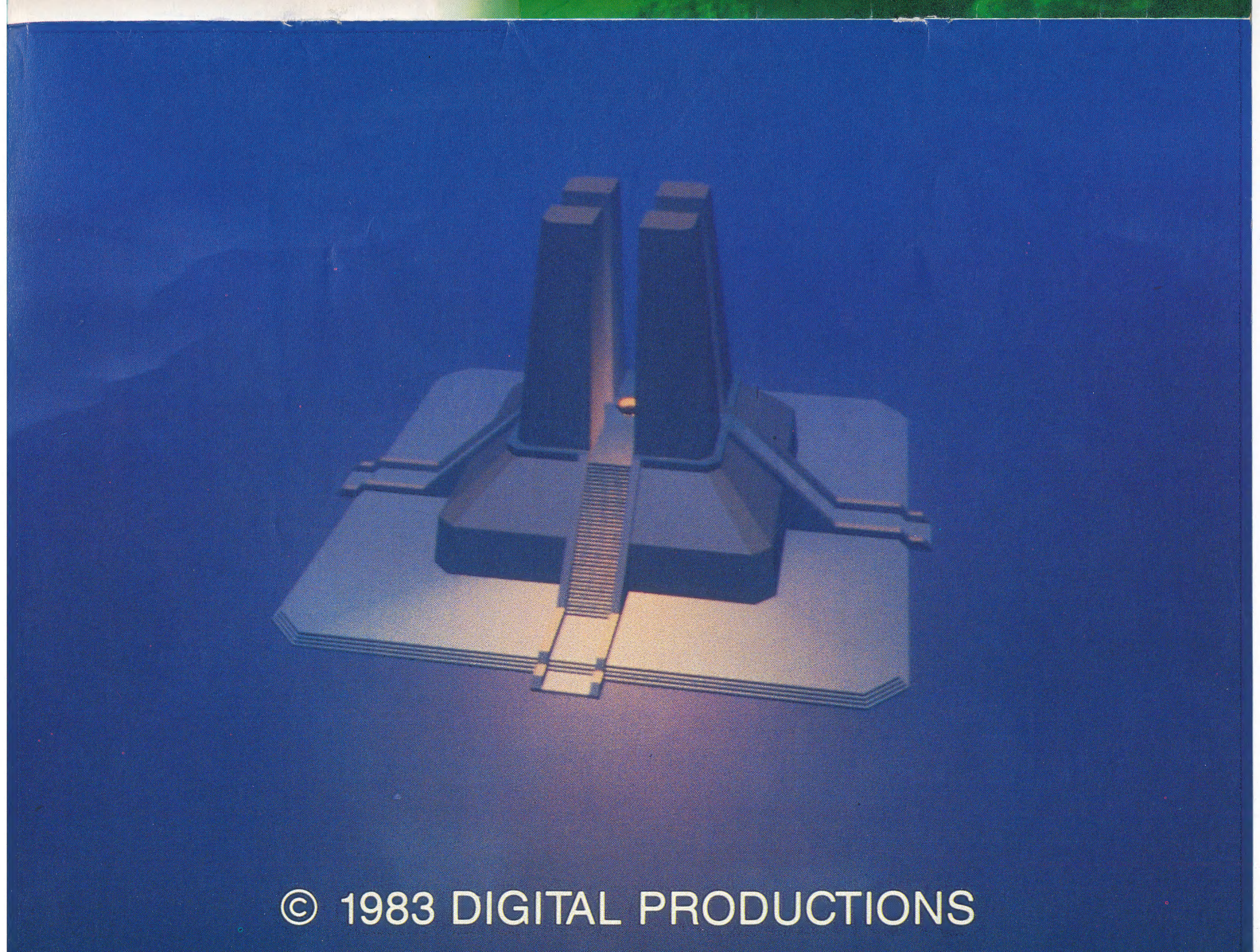
Controlling the ball is fairly straightforward and shooting is performed merely by pressing the fire button. Other available moves include starting and completing a jump shot and spinning 180° to try to get past your opponent. If you are defending, it is possible to go for a steal, try for a rebound or block a shot, though when playing the computer, you'll be hard-pressed to block many. Once on the offensive, however, you'll find the range of moves quite impressive and enough to give the machine a good run for its money.

Imagine's World Series Baseball, referring to the sport's annual league championships in the US, has attempted to incorporate America's enthusiasm for the game. And they've succeeded very well. All of the annoying things about the game — especially the stoppages in play — have been included, and one aspect of the programming — the cheerleaders appearing between innings — is included even though it doesn't actually happen in reality, though most teams do have 'ball girls' picking up the foul balls.

Whether in one- or two-player mode, the game can be played in much the same way as you would in the ballpark. The pitcher has a variety of pitches to choose from and two outfield formations are available. Once on base, the batting team's runner can attempt to steal, although the pitcher can throw to the base he thinks the runner will take.

The screen shows an aerial shot of the playing surface, with some of the crowd in the background. Displayed in the middle of the crowd is a large video screen on which the pitcher is shown in close-up throwing to the batter. It is best to use this display when batting as it gives a much clearer idea of the speed and type of pitch being thrown. Between innings, the line score is displayed. On the bottom-left of the screen is a scoreboard displaying the innings number, strikes (but curiously not balls) and outs. On the bottom right, a board gives the present score of the game.

One of the more entertaining features of the game is the graphics. In themselves, they are not particularly astonishing, but the use of shadows under the players and the ball adds an almost lifelike aspect to the proceedings. As usual, however, in games of this sort, the running motions tend to keep you amused even if you are losing 15-0.


© 1983 DIGITAL PRODUCTIONS